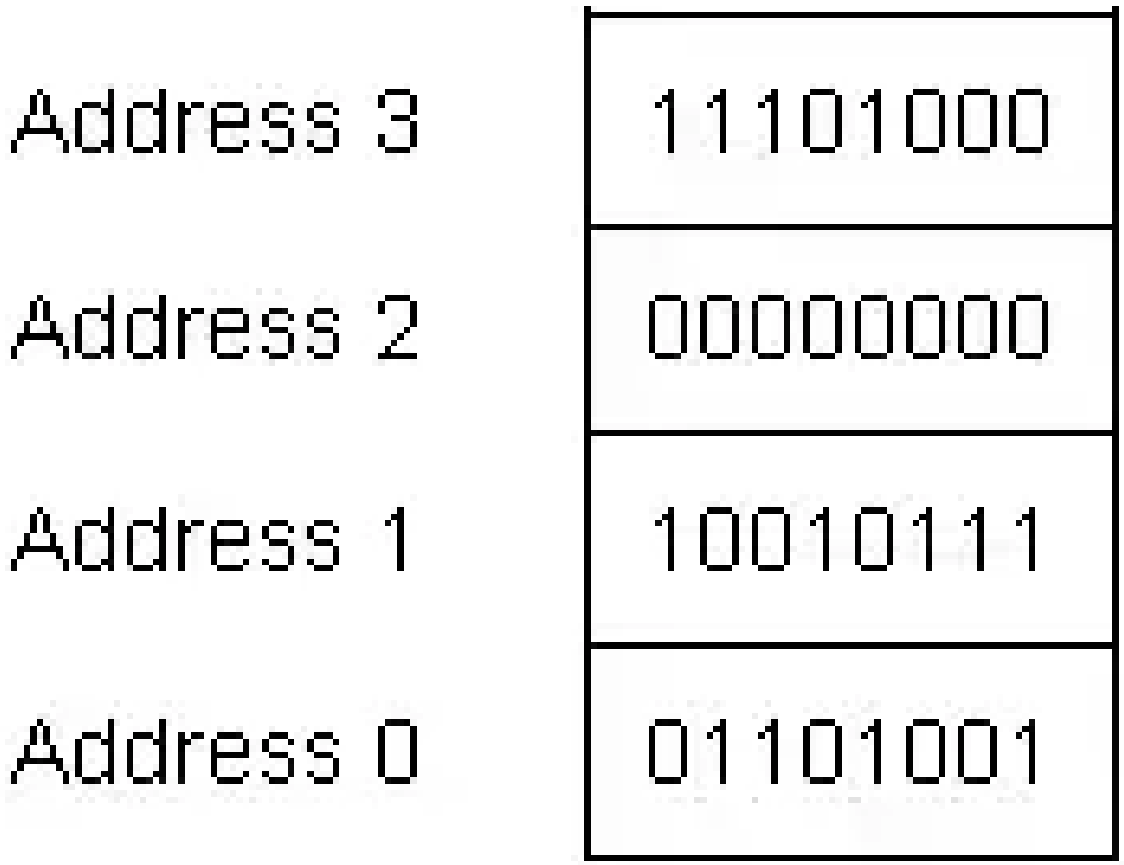
fundamental datatypes

the smallest unit of memory is a **binary digit** (also called a **bit**), which can hold a value of 0 or 1. Memory is organised in sequential units called **memory addresses**(or **addresses** for short). Each bit does not get its own unique memory address as addresses are limited and need to access data bit by bit is rare. Instead each memory address holds a **byte** of data which is comprised of 8 sequential bits.



void is a form of incomplete datatype and basically means no type, a variable cant be defined for void as the system won't know how much memory to allocate to this variable.

```
1  void writeValue(int x) // void here means no return value
2  {
3      std::cout << "The value of x is: " << x << '\n';
4      // no return statement, because this function doesn't return a value
5  }
```

most common use is to define functions that don't return any value.
returning any value in a void function will result in an error.

memory allocation most items will take more than a byte of memory and this memory will vary for different values. We access memory through the variables they are appointed and not the memory addresses they are located in. This helps the compiler hide the amount of exact bytes a object is using and tell us the general amount of memory the object datatype has the capacity to hold.

datatypes and amount of memory they hold

C++ standard does not define the exact size (in bits) for any of the fundamental types.

Category	Type	Minimum Size	Typical Size	Note
Boolean	bool	1 byte	1 byte	
character	char	1 byte	1 byte	always exactly 1 byte
	wchar_t	1 byte	2 or 4 bytes	
	char8_t	1 byte	1 byte	
	char16_t	2 bytes	2 bytes	
	char32_t	4 bytes	4 bytes	
integer	short	2 bytes	2 bytes	
	int	2 bytes	4 bytes	
	long	4 bytes	4 or 8 bytes	
	long long	8 bytes	8 bytes	
floating point	float	4 bytes	4 bytes	
	double	8 bytes	8 bytes	
	long double	8 bytes	8, 12, or 16 bytes	
pointer	std::nullptr_t	4 bytes	4 or 8 bytes	

In order to determine how much data is being held by a datatype in a particular machine we can use **sizeof** operator

```
std::cout << sizeof(bool) <<std::endl ;  //will return one as bool holds 1 byte
```

sizeof() can be used for variables as well

```
int x{};
std::cout << sizeof(x) << std::endl ; // will return 4 as int holds 4 bytes on author's pc
```

signed integers can hold both negative and positive value
signed prefix and int suffix dont change anything so signed short / short / short int are all same datatypes but for redundancy purposes we don't use prefix and suffix and use "short"

short	2 bytes
int	2 bytes / 4 bytes
long	4 bytes
long long	8 bytes

in c++, the signed datatypes allocate on bit as **sign bit** for positive and negative values and remaining bits as **magnitude bits.**
so for a signed integer with n bits of memory , it can store values of range
- 2^{n-1} to **2^{n-1} - 1**
where n-1 is for one bit being sign bit and -1 because of zero being included

overflow is when we operate on a value out of the range for a given datatype and usually results in undefined behaviour.

unsigned Integers only hold positive values and by definition don't have a sign bit so they are capable of storing more positive integers with more bits to work with.
an unsigned integer with n bits of memory can store **0** to **2^n – 1** values
we use unsigned keyword to denote such datatypes

```
unsigned short us{}; //2 bytes
unsigned int{}; // 2/4 bytes
unsigned long{}; // 4 bytes
unsigned long long{}; // 8 bytes
```

if a value more than range is to be stored in unsigned variables then the "out of range" number is divided by maximum upper range limit + one and then the remained is stored.
unsigned Integers are usually avoided because the lower limit 0 is easy to accidentally overflow with.

fixed width integers

size of integral datatypes are not fixed for most because of backwards compatibility with early days of C where systems would often have different performance capabilities and it was better to leave it to compiler to decide the memory allocation for the datatypes ,for example int has minimum size of 2 bytes but is usually 4 bytes on most machines.
To combat this problem we have **fixed width integers** under the #include <cstdint>

std::int8_t	1 byte signed	-128 to 127
std::uint8_t	1 byte unsigned	0 to 255
std::int16_t	2 byte signed	-32,768 to 32,767
std::uint16_t	2 byte unsigned	0 to 65,535
std::int32_t	4 byte signed	-2,147,483,648 to 2,147,483,647
std::uint32_t	4 byte unsigned	0 to 4,294,967,295
std::int64_t	8 byte signed	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
std::uint64_t	8 byte unsigned	0 to 18,446,744,073,709,551,615

downsides for fixed width integers include all machine-wide compatibility & speed.
to ensure we are not compromising speed we have fast integers (std::int_fast8/16/32_t), they will pick a certain datatype between int . short , long on their own based on whichever works the fastest on that given system for the amount of bits provided to work with.
Similarly we have least integers to not compromise memory footprint(std::int_least8/16/32_t) which will pick a certain datatype with least amount of data width used for a given number of bits to be stored in a datatype.

its better to avoid 8 bit fixed width integers as compilers can see them as char.

we also have std::size_t under the #include <cstddef> header which have the ability to hold maximum capacity (ex 32bits in 32bit system and 64bits in 64bit system). They are often used for sizes and counts of arrays or other data types and are unsigned for safer size counts. This is the datatype returned when we use sizeof() operator on any datatype so lets say we want to add a certain size to a sizeof() value we obtained, the type will only match if we use size_t for both the identifiers.

floating point numbers

division of integers is generally not plausible and the fractional part gets dropped of (not rounded) unless one of the operands is floating

```cpp
#include <iostream>

int main()
{
    std::cout << 8 / 5 << '\n';
    return 0;
}
```

the result of above program is “1” as the 0.6 is dropped off from memory.
to solve this we have **floating point variables** which can hold the fractional component.
they can hold signed integers and have three types with different memory capacities.

Category	Type	Minimum Size	Typical Size
floating point	float	4 bytes	4 bytes
	double	8 bytes	8 bytes
	long double	8 bytes	8, 12, or 16 bytes

we use f suffix over a number with decimal value to store it as float or it will be assumed to be a double value that needs to be converted to float which can lead to loss of precision which may generate an error

```cpp
int x{5};      // 5 means integer
double y{5.0}; // 5.0 is a floating point literal (no suffix means double type by default)
float z{5.0f}; // 5.0 is a floating point literal, f suffix means float type
```

```cpp
std::cout << 5.0 << '\n';              //5 as by default std::cout wont print fractional part
std::cout << 5.0f << '\n';             //5 as by default .0 is not written in floating values
std::cout << 6.7f << '\n';             //6.7
std::cout << 9876543.21 << '\n';       //9.87653e+06
```

std::cout has a default precision of 6 significant figures and anything around e+06 / e-06 will be converted into scientific notation.

you can initialise values using scientific notation as well

```cpp
double electronCharge { 1.6e-19 }; // charge on an electron is 1.6 x 10^-19
```

precision

Size	Range	Precision
4 bytes	$\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$ and 0.0	6-9 significant digits, typically 7
8 bytes	$\pm 2.23 \times 10^{-308}$ to $\pm 1.80 \times 10^{308}$ and 0.0	15-18 significant digits, typically 16
80-bits (typically uses 12 or 16 bytes)	$\pm 3.36 \times 10^{-4932}$ to $\pm 1.18 \times 10^{4932}$ and 0.0	18-21 significant digits
16 bytes	$\pm 3.36 \times 10^{-4932}$ to $\pm 1.18 \times 10^{4932}$ and 0.0	33-36 significant digits

we can change the number of significant figures we want to show in a certain output by using an `output manipulator` function named `std::setprecision()`.

output manipulators alter how data is output, and are defined in the *iomanip* header.

```
#include <iomanip> // for output manipulator std::setprecision()
#include <iostream>

int main()
{
    std::cout << std::setprecision(17); // show 17 digits of precision
    std::cout << 3.33333333333333333333333333333333333333f <<'\n'; // f suffix means float
    std::cout << 3.33333333333333333333333333333333333333 << '\n'; // no suffix means double

    return 0;
}
```

Outputs:

```
3.3333332538604736
3.3333333333333335
```

The outputs will have the necessary significant integers but that doesn't mean it will be precise and overrule float and double precision capacities. The changing of numbers due to low precision is called **truncation**.

positive and negative infinities (divide by zero) will give results 1.#iNF / -1.#INF
invalid numbers (zero by zero) will give intermediate results like 1.#IND

boolean values

boolean datatype only have two values of true (1) or false (0)

```
bool b1 {true};          //defining boolean identifiers & bool b1 {1}; means same thing
bool b2  {false};
bool b3 {};      //defaults to false
b4 = false;      //can change assigned value
bool b5 { !true } ;      //assigns false to b4

bool b6 {2};    //will generate error as bool only accepts 0 or 1
bool b7 = 2;    //copy initialisation can be used to assign values other than 0 or 1. all no >= 1 are true

bool b5 {false};
std::cout << true << '\n' ;                //same thing as below , prints 1
std::cout << !b5 << '\n' ;                 //same thing as above , prints 1
```

boolalpha can be used to deal with Booleans as true/false rather than 1/0.

```
std::cout << std::boolalpha ;
std::cout << true << '\n';              // prints true
```

inputting values in booltype

```
bool b6{};
std::cin >> b6;         //std::cin only takes values in 0 and 1 & any input other than 1 is zero including true
std::cin >> std::boolalpha;
std::cin >> b6;         //allows user to input case sensitive true as a input as well
```

boolean values can be used to check whether something is true or not

```
// returns true if x and y are equal, false otherwise
bool isEqual(int x, int y)
{
    return x == y; // operator== returns true if x equals y, and false otherwise
```

char datatype

A **character** can be a single letter, number, symbol, or whitespace. It is stored as an integer under the ASCII code point.

65 = A , 90 = Z , 97 = a , 122 = z

initializing chars

```
char ch1{ 'a'};          //char ch1 {97}; also means same thing but is not preferred method.
```

char is always in a **single quotes**
std::cout will output **a** in both the initialization methods whether **a or 97**.

inputting chars

```
std::cout << "Input a keyboard character: ";

char ch{};
std::cin >> ch;
std::cout << "You entered: " << ch << '\n';
```

std::cin will let you input various characters but the char variable will only hold the first character while remaining user input is left in **input buffer** which can be using subsequent std::cin calls.

If you input abcd via std::cin in a char variable, It will store only "a" but if you add another std:cin to the line of code, it will automatically take b from the input buffer without asking the user for more input.

---

char size, range, and default sign

char is defined by C++ to always be 1 byte in size. By default, a char may be signed or unsigned (though it's usually signed). if you're using chars to hold ASCII characters, you don't need to specify a sign (since both signed and unsigned chars can hold values between 0 and 127).

if you're using a char to hold small integers (something you should not do unless you're explicitly optimizing for space), you should always specify whether it is signed or unsigned. a signed char can hold a number between -128 and 127. an unsigned char can hold a number between 0 and 255.

escape sequences

some sequence of characters in c++ have special meaning. they start with a \.

standalone escape sequences use single quotes '\n' but they can be used in double quotes as well

"first line \n second line"

Name	Symbol	Meaning
Alert	\a	Makes an alert, such as a beep
Backspace	\b	Moves the cursor back one space
Formfeed	\f	Moves the cursor to next logical page
Newline	\n	Moves cursor to next line
Carriage return	\r	Moves cursor to beginning of line
Horizontal tab	\t	Prints a horizontal tab
Vertical tab	\v	Prints a vertical tab
Single quote	\'	Prints a single quote
Double quote	\"	Prints a double quote
Backslash	\\	Prints a backslash.
Question mark	\?	Prints a question mark. No longer relevant. You can use question marks unescaped.
Octal number	\(number)	Translates into char represented by octal
Hex number	\x(number)	Translates into char represented by hex number

examples -

```
#include <iostream>

int main()
{
    std::cout << "\"This is quoted text\"\n";
    std::cout << "This string contains a single backslash \\\n";
    std::cout << "6F in hex is char '\x6F'\n";
    return 0;
}
```

```
"This is quoted text"
This string contains a single backslash \
6F in hex is char 'o'
```

std::cin and handling invalid input

you should always consider how users will (unintentionally or otherwise) misuse your programs. A well-written program will anticipate how users will misuse it, and either handle those cases gracefully or prevent them from happening in the first place (if possible). A program that handles error cases well is said to be **robust**.

when the user enters input in response to an extraction operation(**>>**), that data is placed in a buffer inside of std::cin. A **buffer** (also called a data buffer) is simply a piece of memory set aside for storing data temporarily while it's moved from one place to another. In this case, the buffer is used to hold user input while it's waiting to be extracted to variables.

- If there is data already in the input buffer, that data is used for extraction.
- If the input buffer contains no data, the user is asked to input data for extraction (this is the case most of the time). When the user hits enter, a '\n' character will be placed in the input buffer.
- operator>> extracts as much data from the input buffer as it can into the variable (ignoring any leading whitespace characters, such as spaces, tabs, or '\n').
- Any data that can not be extracted is left in the input buffer for the next extraction.

Example Problem

```
#include <iostream>

double getDouble()
{
    std::cout << "Enter a decimal number: ";
    double x{};
    std::cin >> x;
    return x;
}

char getOperator()
{
    std::cout << "Enter one of the following: +, -, *, or /: ";
    char op{};
    std::cin >> op;
    return op;
}

void printResult(double x, char operation, double y)
{
    std::cout << x << ' ' << operation << ' ' << y << " is ";

    switch (operation)
    {
    case '+':
        std::cout << x + y << '\n';
        return;
    case '-':
        std::cout << x - y << '\n';
        return;
    case '*':
        std::cout << x * y << '\n';
        return;
    case '/':
        std::cout << x / y << '\n';
        return;
    }
}
int main()
{
    double x{ getDouble() };
    char operation{ getOperator() };
    double y{ getDouble() };

    printResult(x, operation, y);

    return 0;
}
```

types of invalid text input

error case 1: extraction succeeds but input is meaningless

1. Check whether the user's input was what you were expecting.
2. If so, return the value to the caller.
3. If not, tell the user something went wrong and have them try again.

```
char getOperator()
{
    while (true) // Loop until user enters a valid input
    {
        std::cout << "Enter one of the following: +, -, *, or /: ";
        char operation{};
        std::cin >> operation;

        // Check whether the user entered meaningful input
        switch (operation)
        {
        case '+':
        case '-':
        case '*':
        case '/':
            return operation; // return it to the caller
        default: // otherwise tell the user what went wrong
            std::cout << "Oops, that input is invalid.  Please try again.\n";
        }
    } // and try again
}
```

error case 2: extraction succeeds but with extraneous input

in case the user inputs extra characters after the needed character, its best to create a method to ignore and remove all these **extratenous** characters.

```
std::cin.ignore(100, '\n');  // clear up to 100 characters out of the buffer, or until a '\n' character is removed
```

this call would remove up to 100 characters, but if the user entered more than 100 characters we'll get messy output again. To ignore all characters up to the next '\n',

```
std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
```

we can also wrap this code line in a function as it would be tedious to write again and again.

```
#include <limits> // for std::numeric_limits

void ignoreLine()
{
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
}
double getDouble()
{
    std::cout << "Enter a decimal number: ";
    double x{};
    std::cin >> x;

    ignoreLine();
    return x;
}
```

this will clear all unnecessary characters in input buffer

error case 3: extraction fails

when the user enters ‘a’ inside the operation variable,Two things will happen at this point: ‘a’ is left in the buffer, and std::cin goes into “failure mode”.

once in “failure mode”, future requests for input extraction will silently fail. Thus in our calculator program, the output prompts still print, but any requests for further extraction are ignored. This means that instead waiting for us to enter an operation, the input prompt is skipped, and we get stuck in an infinite loop because there is no way to reach one of the valid cases.

```
if (!std::cin) // If the previous extraction failed
{
    // Let's handle the failure
    std::cin.clear(); // Put us back in 'normal' operation mode
    ignoreLine();     // And remove the bad input
}
```

in case the user inputs a value that is more or less than the range of the datatype of the variable, there will be overflow. we can solve this as well by the above method.

on Unix systems, entering an end-of-file (EOF) character (via ctrl-D) closes the input stream. This is something that `std::cin.clear()` can’t fix, so `std::cin` never leaves failure mode To handle this case more elegantly, you can explicitly test for EOF using `std::cin.eof()`.

```
// returns true if extraction failed, false otherwise
bool clearFailedExtraction()
{
    // Check for failed extraction
    if (!std::cin) // If the previous extraction failed
    {
        if (std::cin.eof()) // If the stream was closed
        {
            exit(0); // Shut down the program now
        }

        // Let's handle the failure
        std::cin.clear(); // Put us back in 'normal' operation mode
        ignoreLine();     // And remove the bad input

        return true;
    }

    return false;
}
```

putting it all together

```
#include <cstdlib> // for std::exit
#include <iostream>
#include <limits> // for std::numeric_limits

void ignoreLine()
{
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
}

// returns true if extraction failed, false otherwise
bool clearFailedExtraction()
{
    // Check for failed extraction
    if (!std::cin) // If the previous extraction failed
    {
        if (std::cin.eof()) // If the stream was closed
        {
            exit(0); // Shut down the program now
        }

        // Let's handle the failure
        std::cin.clear(); // Put us back in 'normal' operation mode
        ignoreLine();     // And remove the bad input

        return true;
    }

    return false;
}

double getDouble()
{
    while (true) // Loop until user enters a valid input
    {
        std::cout << "Enter a decimal number: ";
        double x{};
        std::cin >> x;

        if (clearFailedExtraction())
        {
            std::cout << "Oops, that input is invalid.  Please try again.\n";
            continue;
        }

        ignoreLine(); // Remove any extraneous input
        return x;     // Return the value we extracted
    }
}

char getOperator()
{
    while (true) // Loop until user enters a valid input
    {
        std::cout << "Enter one of the following: +, -, *, or /: ";
        char operation{};
        std::cin >> operation;

        clearFailedExtraction(); // we'll handle error messaging if extraction failed below
        ignoreLine(); // remove any extraneous input

        // Check whether the user entered meaningful input
        switch (operation)
        {
        case '+':
        case '-':
        case '*':
        case '/':
            return operation; // Return the entered char to the caller
        default: // Otherwise tell the user what went wrong
            std::cout << "Oops, that input is invalid.  Please try again.\n";
        }
    }
}

void printResult(double x, char operation, double y)
{
    std::cout << x << ' ' << operation << ' ' << y << " is ";

    switch (operation)
    {
    case '+':
        std::cout << x + y << '\n';
        return;
    case '-':
        std::cout << x - y << '\n';
        return;
    case '*':
        std::cout << x * y << '\n';
        return;
    case '/':
        std::cout << x / y << '\n';
        return;
    default:
        std::cout << "???";  // Being robust means handling unexpected parameters as well, even though getOperator()
guarantees operation is valid in this particular program
        return;
    }
}

int main()
{
    double x{ getDouble() };
    char operation{ getOperator() };
    double y{ getDouble() };

    printResult(x, operation, y);

    return 0;
}
```

implicit type conversion

in most cases of a function parameter and caller input not matching in datatype, C++ will allow us to convert values of one fundamental type to another fundamental type. The process of converting a value from one type to another type is called **type conversion.**

```cpp
#include <iostream>

void print(double x) // print takes a double parameter
{
    std::cout << x << '\n';
}

int main()
{
    int y { 5 };
    print(y); // y is of type int

    return 0;
}
```

the value held by int variable y(5) will be converted to double value `5.0`, and then copied into parameter x. Here the variable “y” is not becoming a double type variable. It is still a int variable with value “5” but the input it is providing to the function is converted to double for usage.

```cpp
#include <iostream>

void print(int x) // print now takes an int parameter
{
    std::cout << x << '\n';
}

int main()
{
    print(5.5); // warning: we're passing in a double value

    return 0;
}
```

similarly here 5.5 is converted ----> to 5 and the fractional part is dropped in int datatype.

```cpp
int x { 3.5 }; // brace-initialization disallows conversions that result in data loss
```

---

even though the compiler knows how to convert a `double` value to an `int` value, such conversions are disallowed when using brace-initialization.

the standard conversions

the C++ language standard defines how different fundamental types (and in some cases, compound types) can be converted to other types. These conversion rules are called the **standard conversions**.

1) Numeric promotions

2) Numeric conversions

3) Arithmetic conversions

numeric promotion

a 32-bit computer will typically be able to process 32-bits of data at a time. In such cases, an `int` would likely be set to a width of 32-bits, since this is the “natural” size of the data that the CPU operates on (and likely to be the most performant). But what happens when we want our 32-bit CPU to modify an 8-bit value (such as a `char`) or a 16-bit value? Some 32-bit processors (such as 32-bit x86 CPUs) can manipulate 8-bit or 16-bit values directly. However, doing so is often slower than manipulating 32-bit values!

to help address this challenge, C++ defines a category of type conversions informally called the `numeric promotions`. A **numeric promotion** is the type conversion of certain narrower numeric types (such as a `char`) to certain wider numeric types (typically `int` or `double`) that can be processed efficiently. This is done **implicitly**.

```
void printDouble(double d)
{
    std::cout << d << '\n';
}

int main()
{
    printDouble(5.0); // no conversion necessary
    printDouble(4.0f); // numeric promotion of float to double

    return 0;
}
```

```
void printInt(int x)
{
    std::cout << x << '\n';
}

int main()
{
    printInt(2);

    short s{ 3 }; // there is no short literal suffix, so we'll use a variable for this one
    printInt(s); // numeric promotion of short to int

    printInt('a'); // numeric promotion of char to int
    printInt(true); // numeric promotion of bool to int

    return 0;
}
```

numeric conversions

numeric conversions are done **explicitly** on demand

1. Converting an integral type to any other integral type (excluding integral promotions):

```
short s = 3; // convert int to short
long l = 3; // convert int to long
char ch = s; // convert short to char
unsigned int u = 3; // convert int to unsigned int
```

2. Converting a floating point type to any other floating point type (excluding floating point promotions):

```
float f = 3.0; // convert double to float
long double ld = 3.0; // convert double to long double
```

3. Converting a floating point type to any integral type:

```
int i = 3.5; // convert double to int
```

4. Converting an integral type to any floating point type:

```
double d = 3; // convert int to double
```

5. Converting an integral type or a floating point type to a bool:

```
bool b1 = 3; // convert int to bool
bool b2 = 3.0; // convert double to bool
```

arithmetic conversions

the following operators require their operands to be of the same type:

- The binary arithmetic operators: +, -, *, /, %
- The binary relational operators: <, >, <=, >=, ==, !=
- The binary bitwise arithmetic operators: &, ^, |
- The conditional operator ?: (excluding the condition, which is expected to be of type `bool`)

hence if two operands are of different type, one would have to be converted into another for further process.

the compiler has a ranked list of types that looks something like this:

- long double (highest rank)
- double
- float
- long long
- long
- int (lowest rank)

if two values of type `short`: are added, both operands will be converted to int and added as short is not on priority list.

explicit type conversion

```
1 | double d = 10 / 4; // does integer division, initializes d with value 2.0
```

because `10` and `4` are both of type `int`, integer division is performed, and the expression evaluates to `int` value `2` Here we can just use their floating forms (10.0 and 4.0) to get the result but for variables instead of literals, we would need a forced explicit type conversion.

fortunately, C++ comes with a number of different **type casting operators** (more commonly called **casts**) that can be used by the programmer to request that the compiler perform a type conversion.

C++ supports 5 different types of casts: `C-style casts`, `static casts`, `const casts`, `dynamic casts`, and `reinterpret casts`. The latter four are sometimes referred to as **named casts**.

C-style casts

```
int x { 10 };
int y { 4 };
```

```
double d { (double)x / y }; // convert x to a double so we get floating point division
```

the left operand of operator/ now evaluates to a floating point value, the right operand will be converted to a floating point value as well, and the division will be done using floating point division instead of integer division!

C-style casts are just a type name, parenthesis, and variable or value, they are both difficult to identify (making your code harder to read) and even more difficult to search for.

static_casts

the static_cast operator takes an expression as input, and returns the evaluated value converted to the type specified inside the angled brackets. static_cast is best used to convert one fundamental type into another. static_cast provides compile-time type checking, making it harder to make an inadvertent error. static_cast is also (intentionally) less powerful than C-style casts, so you can't inadvertently remove const or do other things you may not have intended to do.

```
convertedvariable = static_cast<datatype>(value);              //converts value to desired datatype
std::cout << charvariable << "has value" << static_cast<int>(charvariable);       //to print char values as int counterparts
```

static_cast can also return overflowing unsigned int to signed form.

narrowing conversions

a **narrowing conversion** is a potentially unsafe numeric conversion where the destination type may not be able to hold all the values of the source type like float to integral. Narrowing conversions should be avoided as much as possible, because they are potentially unsafe, and thus a source of potential errors. implicit narrowing conversions will result in compiler warnings, with the exception of signed/unsigned conversions and in unavoidable cases, static_cast must be used.

constants

named constants are constant values that are associated with an identifier. These are also sometimes called **symbolic constants**, or occasionally just **constants**. However, there are many cases where it is useful to define variables with values that cannot be changed.

```
  const double gravity; // error: const variables must be initialized
  gravity = 9.9;        // error: const variables can not be changed
```

although it is a well-known oxymoron, a variable whose value cannot be changed is called a **constant variable**.

```
std::cout << "Enter your age: ";
int age{};
std::cin >> age;

const int constAge { age }; // initialize const variable using non-const value

age = 5;      // ok: age is non-const, so we can change its value
constAge = 6; // error: constAge is const, so we cannot change its value

return 0;
```

don't use const when passing by value **void printInt(const int x)** because the function parameter ought to change anyways or returning by value **const int getValue()** as its simply ignored.

prefer constant variables over pre-processor macros as constants have defined scope with it being global if outside any function and local if defined inside a function.

compile Time evaluation

he **as-if rule** says that the compiler can modify a program however it likes in order to produce more optimized code, so long as those modifications do not affect a program's "observable behavior".

for example : int x { 3+4}; will be replaced by int x {7}; before the code even runs.
we also use this evaluation on constant expressions which only contain compile time constants and operators/functions supporting compile time evaluation.

```
    // Non-const variables:
    int a { 5 };                // 5 is a constant expression
    double b { 1.2 + 3.4 };     // 1.2 + 3.4 is a constant expression

    // Const integral variables with a constant expression initializer
    // are compile-time constants:
    const int c { 5 };          // 5 is a constant expression
    const int d { c };          // c is a constant expression
    const long e { c + 2 };     // c + 2 is a constant expression

    // Other const variables are runtime constants:
    const int f { a };          // a is not a constant expression
    const int g { a + 1 };      // a + 1 is not a constant expression
    const long h { a + c };     // a + c is not a constant expression
    const int i { getNumber() }; // getNumber() is not a constant expression

    const double j { b };       // b is not a constant expression
    const double k { 1.2 };     // 1.2 is a constant expression
```

an expression that is not a constant expression is sometimes called a **runtime expression**. For example, `std::cout << x << '\n'` is a runtime expression, both because `x` is not a compile-time constant, and because `operator<<` doesn't support compile-time evaluation when used for output (since output can't be done at compile-time).

this includes optimizations of certain variables like

```
#include <iostream>

int main()
{
    int x { 7 };             // x is non-const
    std::cout << x << '\n'; // x is a non-constant subexpression

    return 0;
}
```

even though `x` is non-const, a smart compiler might realize that `x` will always evaluate to `7` in this particular program and replace x with 7 in print statement and then remove initialiser for x since its no longer used anywhere else in the program. needless to say, if the initialiser for x was done with const instead of int datatype, it would be easier to optimise since you can't change the value for const variables.

we can use **constexpr** instead of **const** for initialisation of compile time constant variables

```
constexpr double gravity { 9.8 }; // ok: 9.8 is a constant expression
constexpr int sum { 4 + 5 };      // ok: 4 + 5 is a constant expression
constexpr int something { sum };  // ok: sum is a constant expression

std::cout << "Enter your age: ";
int age{};
std::cin >> age;

constexpr int myAge { age };      // compile error: age is not a constant expression
constexpr int f { five() };       // compile error: return value of five() is not a constant expression
```

consteval functions are compiled at compile time whereas constexpr function can be compiled at compile time based on context of the function.

type aliases

In C++, we can create aliases for datatypes using the **using** keyword.

```
using Distance = double; // define Distance as an alias for type double
```

```
Distance milesToDestination{ 3.4 }; // defines a variable of type double
```

in modern C++, the convention is to name type aliases (or any other type) that you define yourself starting with a capital letter, and using no suffix. The capital letter helps differentiate the names of types from the names of variables and functions (which start with a lower case letter), and prevents naming collisions between them.

When using this naming convention, it is common to see this usage:

```
1 | void printDistance(Distance distance); // Distance is some defined type
```

In this case, `Distance` is the type, and `distance` is the parameter name. C++ is case-sensitive, so this is fine.

```
using Miles = long; // define Miles as an alias for type long
using Speed = long; // define Speed as an alias for type long

Miles distance { 5 }; // distance is actually just a long
Speed mhz  { 3200 };  // mhz is actually just a long

// The following is syntactically valid (but semantically meaningless)
distance = mhz;

return 0;
```

although conceptually we intend `Miles` and `Speed` to have distinct meanings, both are just aliases for type `long`. This effectively means `Miles`, `Speed`, and `long` can all be used interchangeably. And indeed, when we assign a value of type `Speed` to a variable of type `Miles`, the compiler only sees that we're assigning a value of type `long` to a variable of type `long`, and it will not complain.

because the compiler does not prevent these kinds of semantic errors for type aliases, we say that aliases are not **type safe**. In spite of that, they are still useful.

a type alias defined inside a block has block scope and is usable only within that block, whereas a type alias defined in the global namespace has global scope and is usable to the end of the file.

typedef is a older way of typealiasing.

```
// The following aliases are identical
typedef long Miles;
using Miles = long;
```

type aliases should be used primarily in cases where there is a clear benefit to code readability or code maintenance. This is as much of an art as a science. Type aliases are most useful when they can be used in many places throughout your code, rather than in fewer places.

type deduction

In C++, we are required to provide an explicit type for all objects but if the literal and type are both same then it is redundant to provide the same information twice to the compiler.

```
double d{ 5.0 };
```

Here double and 5.0, both are telling the same information. **Type deduction** (also sometimes called **type inference**) is a feature that allows the compiler to deduce the type of an object from the object's initializer with **auto** keyword.

```
auto d{ 5.0 }; // 5.0 is a double literal, so d will be type double
auto i{ 1 + 2 }; // 1 + 2 evaluates to an int, so i will be type int
auto x { i }; // i is an int, so x will be type int too
```

const is dropped in auto type deduction

```
const int x { 5 }; // x has type const int
auto y { x };      // y will be type int (const is dropped)
```

we must supply const ourselves by constexpr auto y{x};

For historical reasons, string literals in C++ have a strange type.

```
auto s { "Hello, world" }; // s will be type const char*, not std::string
```

To convert string literals into std::string or std::stringveiw, we use s and sv literals.

```
auto s1 { "goo"s };  // "goo"s is a std::string literal, so s1 will be deduced as a std::string
auto s2 { "moo"sv }; // "moo"sv is a std::string_view literal, so s2 will be deduced as a std::string_view
```

Functions can also use type deduction.

```
auto add(int x, int y)
{
    return x + y;
}
```

provided that all return statements return values of same type, or static_cast must be used

A major downside of functions that use an `auto` return type is that such **functions must be fully defined before they can be used** (a forward declaration is not sufficient).

we can also use auto keyword with **trailing return syntax**

```
auto add(int x, int y) -> int;
auto divide(double x, double y) -> double;
auto printSomething() -> void;
auto generateSubstring(const std::string &s, int start, int len) -> std::string;
```

here auto isn't performing type deduction as return type is provided and is mostly used for readability purposes and for some other complex features like lambdas(anonymous functions)

auto keyword also **can't be used as function parameters**

This would only compile in C++20 but for a different reason (function templates, not type deductions)

Overall, the modern consensus is that type deduction is generally safe to use for objects, and that doing so can help make your code more readable by de-emphasizing type information so the logic of your code stands out better.

sharing global constants across multiple files

method 1
define all global constants in a namespace of a headerfile and include the headerfile in cpp files you want to use the constants via the scope resolution operator.

constants.h:

```cpp
#ifndef CONSTANTS_H
#define CONSTANTS_H

// define your own namespace to hold constants
namespace constants
{
    // constants have internal linkage by default
    constexpr double pi { 3.14159 };
    constexpr double avogadro { 6.0221413e23 };
    constexpr double myGravity { 9.2 }; // m/s^2 -- gravity is light on this planet
    // ... other related constants
}
#endif
```

main.cpp:

```cpp
#include "constants.h" // include a copy of each constant in this file

#include <iostream>

int main()
{
    std::cout << "Enter a radius: ";
    double radius{};
    std::cin >> radius;

    std::cout << "The circumference is: " << 2 * radius * constants::pi << '\n';

    return 0;
}
```

while this is simple (and fine for smaller programs), every time constants.h gets #included into a different code file, each of these variables is copied into the including code file. Therefore, if constants.h gets included into 20 different code files, each of these variables is duplicated 20 times. Header guards won't stop this from happening. Changing a single constant value would require recompiling every file that includes the constants header, which can lead to lengthy rebuild times for larger projects. If the constants are large in size and can't be optimized away, this can use a lot of memory.

method 2

One way to solve this would be make the variables external, so instead of including them in every file and recompiling each file, the identifiers would be accessible in all files with forward declarations while only needing recompiling only one file in case of a change.

constants.cpp:

```cpp
#include "constants.h"

namespace constants
{
    // actual global variables
    extern constexpr double pi { 3.14159 };
    extern constexpr double avogadro { 6.0221413e23 };
    extern constexpr double myGravity { 9.2 }; // m/s^2 -- gravity is light on this planet
}
```

constants.h:

```
#ifndef CONSTANTS_H
#define CONSTANTS_H

namespace constants
{
    // since the actual variables are inside a namespace, the forward declarations need to be inside a namespace as well
    // we can't forward declare variables as constexpr, but we can forward declare them as (runtime) const
    extern const double pi;
    extern const double avogadro;
    extern const double myGravity;
}

#endif
```

main.cpp:

```
#include "constants.h" // include all the forward declarations

#include <iostream>

int main()
{
    std::cout << "Enter a radius: ";
    double radius{};
    std::cin >> radius;

    std::cout << "The circumference is: " << 2 * radius * constants::pi << '\n';

    return 0;
}
```

however, there are a couple of downsides to this method. First, these constants are now considered compile-time constants only within the file they are actually defined in (`constants.cpp`). In other files, the compiler will only see the forward declaration, which doesn't define a constexpr value (and must be resolved by the linker). This means in other files, these are treated as runtime constant values, not compile-time constants. Thus outside of `constants.cpp`, these variables can't be used anywhere that requires a compile-time constant. Second, because compile-time constants can typically be optimized more than runtime constants, the compiler may not be able to optimize these as much.

method 3

we can also use **inline variables** to prevent the constants from being uniquely copied into each codefile and be instantiated once and shared across all files

constants.h:

```
#ifndef CONSTANTS_H
#define CONSTANTS_H

// define your own namespace to hold constants
namespace constants
{
    inline constexpr double pi { 3.14159 }; // note: now inline constexpr
    inline constexpr double avogadro { 6.0221413e23 };
    inline constexpr double myGravity { 9.2 }; // m/s^2 -- gravity is light on this planet
    // ... other related constants
}
#endif
```

main.cpp:

```
#include "constants.h"

#include <iostream>

int main()
{
    std::cout << "Enter a radius: ";
    double radius{};
    std::cin >> radius;

    std::cout << "The circumference is: " << 2 * radius * constants::pi << '\n';

    return 0;
}
```

this method does retain the downside of requiring every file that includes the constants header be recompiled if any constant value is changed.

const and constexpr global variables have **internal linkage** which means they can't be accessed by other files even via linker whereas non const global variables have **external linkage.**

we can use the **static** keyword to **make non const globals have internal linkage** (ex. static int g_x {1};). even functions, who have external linkage by default due to forward declarations, can be made **internally linked** by static keyword. (ex. static int add(int x, int y);) which would make them inaccessible to other files. This is helpful to avoid naming collisions as they won't be linked by linker.

add.cpp:

```
// This function is declared as static, and can now be used only within this file
// Attempts to access it from another file via a function forward declaration will fail
[[maybe_unused]] static int add(int x, int y)
{
    return x + y;
}
```

main.cpp:

```
#include <iostream>

static int add(int x, int y); // forward declaration for function add

int main()
{
    std::cout << add(3, 4) << '\n';

    return 0;
}
```

This program won't link, because function `add` is not accessible outside of `add.cpp`.

---

using the static keyword on a local variable changes its duration from automatic duration to static duration. This means the variable is now created at the start of the program, and destroyed at the end of the program (just like a global variable).
automatic duration means they are created at the point of definition, and destroyed when the block is exited.

```
#include <iostream>

void incrementAndPrint()
{
    int value{ 1 }; // automatic duration by default
    ++value;
    std::cout << value << '\n';
} // value is destroyed here

int main()
{
    incrementAndPrint();
    incrementAndPrint();
    incrementAndPrint();

    return 0;
}
```

here the functions call will print 2,2,2 but if the datatype of value in function was static int, the function calls would print 2,3,4 instead because the value of "value" is retained even when the block ends because static duration lasts throughout the program instead of automatic duration for a local variable that destroys at the end of the code block.

ID Generation

static keyword is used for ID generation since the value is retained even after function ends, a simple increment code is enough to assign multiple consecutive values to similar yet different objects like zombies in a video game. This is incredibly helpful in identifying which similar yet different object is causing the bug in codes.

```
int generateID()
{
    static int s_itemID{ 0 };
    return s_itemID++; // makes copy of s_itemID, increments the real s_itemID, then returns the value in the copy
}
```

Each subsequent function call will return a incremented ID.

extern keyword

to make a global variable external (and thus accessible by other files), we can use the `extern` keyword during their initialisation. (ex. extern const int g_x {}; extern constexpr int g_x {};)

a.cpp:

```
1  // global variable definitions
2  int g_x { 2 }; // non-constant globals have external linkage by default
3  extern const int g_y { 3 }; // this extern gives g_y external linkage
```

main.cpp:

```
1  #include <iostream>
2
3  extern int g_x; // this extern is a forward declaration of a variable named g_x that is defined somewhere else
4  extern const int g_y; // this extern is a forward declaration of a const variable named g_y that is defined somewhere
   else
5
6  int main()
7  {
8      std::cout << g_x << ' ' << g_y << '\n'; // prints 2 3
9
10     return 0;
11 }
```

we would still need extern forward declaration to access the identifiers from other files to differentiate between uninitialized variable declaration and called variable from other file.

Warning

If you want to define an uninitialized non-const global variable, do not use the extern keyword, otherwise C++ will think you're trying to make a forward declaration for the variable.

Warning

Although constexpr variables can be given external linkage via the `extern` keyword, they can not be forward declared as constexpr. This is because the compiler needs to know the value of the constexpr variable (at compile time). If that value is defined in some other file, the compiler has no visibility on what value was defined in that other file.

However, you can forward declare a constexpr variable as const, which the compiler will treat as a runtime const. This isn't particularly useful.

why (non-const) global variables are evil

If you were to ask a veteran programmer for *one* piece of advice on good programming practices, after some thought, the most likely answer would be, “Avoid global variables!”. And with good reason: global variables are one of the most historically abused concepts in the language. Although they may seem harmless in small academic programs, they are often problematic in larger ones. When developers tell you that global variables are evil, they’re usually not talking about *all* global variables. They’re mostly talking about non-const global variables.

By far the biggest reason non-const global variables are dangerous is because their values can be changed by *any* function that is called, and there is no easy way for the programmer to know that this will happen. One of the key reasons to declare local variables as close to where they are used as possible is because doing so minimizes the amount of code you need to look through to understand what the variable does. Global variables are at the opposite end of the spectrum -- because they can be accessed anywhere, you might have to look through the entire program to understand their usage. In small programs, this might not be an issue. In large ones, it will be.

As a rule of thumb, any use of a global variable should meet at least the following two criteria: There should only ever be one of the thing the variable represents in your program, and its use should be ubiquitous throughout your program.

Many new programmers make the mistake of thinking that something can be implemented as a global because only one is needed $right\ now$. For example, you might think that because you’re implementing a single player game, you only need one player. But what happens later when you want to add a multiplayer mode (versus or hotseat)?

inline functions and variables

every function call requires that the address of current instructions are stored as to know where to return to while the execution path jumps to the code where function is written while the parameters are initialised along with the return value. Fortunately, the C++ compiler has a trick that it can use to avoid such overhead cost: **Inline expansion** is a process where a function call is replaced by the code from the called function's definition.

```
inline int min(int x, int y) // inline keyword means this function is an inline function
{
    return (x < y) ? x : y;
}
```

```
#include <iostream>

int main()
{
    std::cout << ((5 < 6) ? 5 : 6) << '\n';
    std::cout << ((3 < 2) ? 3 : 2) << '\n';
    return 0;
}
```

beyond removing the cost of function call, inline expansion can also allow the compiler to optimize the resulting code more efficiently -- for example, because the expression `((5 < 6) ? 5 : 6)` is now a constant expression, the compiler could further optimize the first statement in `main()` to `std::cout << 5 << '\n';`.

the decision about whether a function would benefit from being made inline (because removal of the function call overhead outweighs the cost of a larger executable) is not straightforward. Inline expansion could result in performance improvements, performance reductions, or no change to performance at all, depending on the relative cost of a function call, the size of the function, and what other optimizations can be performed.

historically, compilers either didn't have the capability to determine whether inline expansion would be beneficial, or were not very good at it. For this reason, C++ provided the keyword `inline`, which was originally intended to be used as a hint to the compiler that a function would (probably) benefit from being expanded inline.

However, in modern C++, the `inline` keyword is no longer used to request that a function be expanded inline. There are quite a few reasons for this:

- Using `inline` to request inline expansion is a form of premature optimization, and misuse could actually harm performance.
- The `inline` keyword is just a hint -- the compiler is completely free to ignore a request to inline a function. This is likely to be the result if you try to inline a lengthy function! The compiler is also free to perform inline expansion of functions that do not use the `inline` keyword as part of its normal set of optimizations.
- The `inline` keyword is defined at the wrong level of granularity. We use the `inline` keyword on a function definition, but inline expansion is actually determined per function call. It may be beneficial to expand some function calls and detrimental to expand others, and there is no syntax to influence this.

prevention of ODR (One Definition Rule)

inline functions prevent ODR provided different definitions across the program have same definition

main.cpp:

```cpp
#include <iostream>

double circumference(double radius); // forward declaration

inline double pi() { return 3.14159; }

int main()
{
    std::cout << pi() << '\n';
    std::cout << circumference(2.0) << '\n';

    return 0;
}
```

math.cpp

```cpp
inline double pi() { return 3.14159; }

double circumference(double radius)
{
    return 2.0 * pi() * radius;
}
```

inline functions are defined in header files where they can be #included into the top of any code file that needs the full definition of the identifier while ensuring that all inline definitions are identical and don't result in error.

pi.h:

```cpp
#ifndef PI_H
#define PI_H

inline double pi() { return 3.14159; }

#endif
```

main.cpp:

```cpp
#include "pi.h" // will include a copy of pi() here
#include <iostream>

double circumference(double radius); // forward declaration

int main()
{
    std::cout << pi() << '\n';
    std::cout << circumference(2.0) << '\n';

    return 0;
}
```

math.cpp

```cpp
#include "pi.h" // will include a copy of pi() here

double circumference(double radius)
{
    return 2.0 * pi() * radius;
}
```

Why not make all functions inline and defined in a header file?

There are a few good reasons:

- It can increase your compile times. When a function in a code file changes, only that code file needs to be recompiled. When an inline function in a header file changes, every code file that includes that header (either directly or via another header) needs to recompiled. On large projects, this can have a drastic impact.
- It can lead to more naming collisions since you'll end up with more code in a single code file.

C++17 introduces **inline variables**, which are variables that are allowed to be defined in multiple files. Inline variables work similarly to inline functions, and have the same requirements (the compiler must be able to see an identical full definition everywhere the variable is used).

constexpr functions are **implicitly inline**

inlined namespace

an **inline namespace** is a namespace that is typically used to version content. Much like an unnamed namespace, anything declared inside an inline namespace is considered part of the global namespace. However, unlike unnamed namespaces, inline namespaces don't affect linkage. This is usually done for keeping different versions of same name functions as the inlined function in namespace will always be called from global namespace and separate versions when called explicitly

```
#include <iostream>

inline namespace V1 // declare an inline namespace named V1
{
    void doSomething()
    {
        std::cout << "V1\n";
    }
}

namespace V2 // declare a normal namespace named V2
{
    void doSomething()
    {
        std::cout << "V2\n";
    }
}

int main()
{
    V1::doSomething(); // calls the V1 version of doSomething()
    V2::doSomething(); // calls the V2 version of doSomething()

    doSomething(); // calls the inline version of doSomething() (which is V1)

    return 0;
}
```

we can also use inline unnamed namespaces for internally linked global namespace code but its mostly used for clarity and specification.

inlined unnamed namespaces for internally linked global namespace code is legal but mostly used for clarity and specification of codecluster

number systems

there are 4 number systems including decimal, binary, octal and hexadecimal system.

initialising Binary Values to variables

```
int bin{};        // assume 16-bit ints
bin = 0b1;        // assign binary 0000 0000 0000 0001 to the variable
bin = 0b11;       // assign binary 0000 0000 0000 0011 to the variable
bin = 0b1010;     // assign binary 0000 0000 0000 1010 to the variable
bin = 0b11110000; // assign binary 0000 0000 1111 0000 to the variable
```

Decimal	0	1	2	3	4	5	6	7	8	9	10	11
Octal	0	1	2	3	4	5	6	7	10	11	12	13

```
int x{ 012 }; // 0 before the number means this is octal
```

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Hexadecimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11

```
int x{ 0xF }; // 0x before the number means this is hexadecimal
```

conversion from binary/octal/hexadecimal is done by adding the powers with maximum base for each digit and conversion from decimal to binary/octal/hexadecimal is done by breaking the decimal into the powers with maximum base and writing each coefficient.

numbers are always output in decimal unless output format is changed

```
// std::bitset<8> means we want to store 8 bits
std::bitset<8> bin1{ 0b1100'0101 }; // binary literal for binary 1100 0101
std::bitset<8> bin2{ 0xC5 }; // hexadecimal literal for binary 1100 0101

std::cout << bin1 << '\n' << bin2 << '\n';
std::cout << std::bitset<4>{ 0b1010 } << '\n'; // create a temporary std::bitset and print it
```

```
11000101
11000101
1010
```

std::bitset is under #include <bitset> preprocessor
in C++20 and C++23 we have improved methods to output binary numbers

```
1  #include <format> // C++20
2  #include <iostream>
3  #include <print> // C++23
4
5  int main()
6  {
7      std::cout << std::format("{:b}\n", 0b1010);  // C++20
8      std::cout << std::format("{:#b}\n", 0b1010); // C++20
9
10     std::println("{:b} {:#b}", 0b1010, 0b1010);  // C++23
11
```

```
1010
0b1010
1010 0b1010
```

Hexadecimal and Octal Ouputs

```
int x { 12 };
std::cout << x << '\n'; // decimal (by default)
std::cout << std::hex << x << '\n'; // hexadecimal
std::cout << x << '\n'; // now hexadecimal
std::cout << std::oct << x << '\n'; // octal
std::cout << std::dec << x << '\n'; // return to decimal
std::cout << x << '\n'; // decimal
```

literals are values that are inserted directly into the code and cannot be changed. For example 5 will always mean integral value of five.

Literal value	Examples	Default literal type
integer value	5, 0, -3	int
boolean value	true, false	bool
floating point value	1.2, 0.0, 3.4	double (not float!)
character	'a', '\n'	char
C-style string	"Hello, world!"	const char[14]

If the default type of a literal is not as desired, you can change the type of a literal by adding a suffix.

Data type	Suffix	Meaning
integral	u or U	unsigned int
integral	l or L	long
integral	ul, uL, Ul, UL, lu, lU, Lu, LU	unsigned long
integral	ll or LL	long long
integral	ull, uLL, Ull, ULL, llu, llU, LLu, LLU	unsigned long long
integral	z or Z	The signed version of std::size_t (C++23)
integral	uz, uZ, Uz, UZ, zu, zU, Zu, ZU	std::size_t (C++23)
floating point	f or F	float
floating point	l or L	long double
string	s	std::string
string	sv	std::string_view

In most cases, suffixes aren't needed (except for f)

magic numbers
A **magic number** is a literal (usually a number) that either has an unclear meaning or may need to be changed later. In complex programs, it can be very difficult to infer what a literal represents, unless there's a comment to explain it. Using magic numbers is generally considered bad practice because, in addition to not providing context as to what they are being used for, they pose problems if the value needs to change. We will have to change the value or text multiple times throughout the code. This can be easily avoided by using symbolic constants that are named as well for context of their meaning.

string literals

“Hello, world!” in std::cout << “Hello, world!”; is called a string literal (in opposition to char literals which are between single quotes).
Because strings are commonly used in programs, most modern programming languages include a fundamental string data type. For historical reasons, strings are not a fundamental type in C++. Rather, they have a strange, complicated type that is hard to work with. Such strings are often called **C strings** or **C-style strings**, as they are inherited from the C-language.

all C-style string literals have an implicit null terminator. Consider a string such as `"hello"`. While this C-style string appears to only have five characters, it actually has six: `'h'`, `'e'`, `'l'`, `'l'`, `'o'`, and `'\0'` (a character with ASCII code 0). This trailing ‘\0’ character is a special character called a **null terminator**, and it is used to indicate the end of the string. A string that ends with a null terminator is called a **null-terminated string**. This is the reason the string `"Hello, world!"` has type `const char[14]` rather than `const char[13]` -- the hidden null terminator counts as a character.

while C-style string literals are fine to use, C-style string variables behave oddly, are hard to work with (e.g. you can’t use assignment to assign a C-style string variable a new value without using functions like strcpy), and are dangerous (e.g. if you copy a larger C-style string into the space allocated for a shorter C-style string like char shortstring[3] with 2 characters and 1 null character to fill into, undefined behavior will result). In modern C++, C-style string variables are best avoided.

introducing `std::string`

the easiest way to work with strings and string objects in C++ is via the `std::string` type, which lives in the <string> header.

```
#include <iostream>
#include <string>
std::string stringname {};                          //empty string
std::string stringname {“this is a astring”};       //we can directly assign strings
stringname = “this is a new string:};               // we can change string values
std::cout << stringname << ‘\n’ ;                   // we can output such string using std::cout
```

std::strings can store strings of different sizes and it will request additional memory at runtime if the space is not enough to store a large string.

std::cin can only take value till whitespaces in strings, this could be a problem because the surname in case of a name input, could get into the input stream and get inputted in future std::cin inputs.

```
std::getline(std::cin >> std::ws, input);
```

can be used to obtain the whole line with std::ws to ensure all leading whitespace characters are ignored which is important

```
std::cout << "Pick 1 or 2: ";
int choice{};
std::cin >> choice;

std::cout << "Now enter your name: ";
std::string name{};
std::getline(std::cin, name); // note: no std::ws here

std::cout << "Hello, " << name << ", you picked " << choice << '\n';
```

```
Pick 1 or 2: 2
Now enter your name: Hello, , you picked 2
```

this program first asks you to enter 1 or 2, and waits for you to do so. All good so far. Then it will ask you to enter your name. However, it won't actually wait for you to enter your name! Instead, it prints the "Hello" string, and then exits.

extraction operator (>>) ignores leading whitespaces while std::getline() sees it a individual line. When we input a value, the std::cin also captures the '\n' character left behind by enter key. This whitespace characters is left in the input stream and picked by std::getline which was not intended to happen.

the length of a std::string

the number of characters in string can be known by stringname.length() function which will return the total characters **excluding the null character**, even though C++ is null terminated like C. Its syntactically written as object.function() and not function(object) like normal functions because this is a **member function** specially declared inside of std::string.
The value is returned in **size_t type** and we need to use static_cast to assign it to int variables.

initialising std::strings or passing std::strings as value to a function are very expensive on computational time and memory, as a copy of the string is made to complete such tasks.

converting literals to std::string type

```
using namespace std::string_literals; // easy access to the s suffix

std::cout << "foo\n";   // no suffix is a C-style string literal
std::cout << "goo\n"s;  // s suffix is a std::string literal
```

The "s" suffix lives in the namespace `std::literals::string_literals`.

The most concise way to access the literal suffixes is via using-directive `using namespace std::literals`. However, this imports *all* of the standard library literals into the scope of the using-directive, which brings in a bunch of stuff you probably aren't going to use.

We recommend `using namespace std::string_literals`, which imports only the literals for `std::string`.

constexpr std::string stringname{"this is a string"s}; may give compile error because it was only included after C++17

std::string_view

to address the issue with `std::string` being expensive to initialize (or copy), C++17 introduced `std::string_view` (which lives in the <string_view> header). `std::string_view` provides read-only access to an *existing* string (a C-style string, a `std::string`, or another `std::string_view`) without making a copy.

```
#include <iostream>
#include <string_view> // C++17

// str provides read-only access to whatever argument is passed in
void printSV(std::string_view str) // now a std::string_view
{
    std::cout << str << '\n';
}

int main()
{
    std::string_view s{ "Hello, world!" }; // now a std::string_view
    printSV(s);

    return 0;
}
```

when we initialize `std::string_view s` with C-style string literal `"Hello, world!"`, `s` provides read-only access to “Hello, world!” without making a copy of the string. When we pass `s` to `printSV()`, parameter `str` is initialized from `s`. This allows us to access “Hello, world!” through `str`, again without making a copy of the string.

it can be used to **initialise with c style literals, std::strings and std::string_veiws**

```
std::string_view s1 { "Hello, world!" }; // initialize with C-style string literal
std::cout << s1 << '\n';

std::string s{ "Hello, world!" };
std::string_view s2 { s };  // initialize with std::string
std::cout << s2 << '\n';

std::string_view s3 { s2 }; // initialize with std::string_view
std::cout << s3 << '\n';
```

functions with std::string_veiw parameter **can also be called using c-style literals and std::strings**

```
void printSV(std::string_view str)
{
    std::cout << str << '\n';
}

int main()
{
    printSV("Hello, world!"); // call with C-style string literal

    std::string s2{ "Hello, world!" };
    printSV(s2); // call with std::string

    std::string_view s3 { s2 };
    printSV(s3); // call with std::string_view

    return 0;
}
```

we cant pass std::string_veiw argument in function with std::string parameter as std::strings create expensive copies which is not intended with the use of std::string_veiws.

```
void printString(std::string str)
{
    std::cout << str << '\n';
}

int main()
{
    std::string_view sv{ "Hello, world!" };

    // printString(sv);   // compile error: won't implicitly convert std::string_view to a std::string

    std::string s{ sv }; // okay: we can create std::string using std::string_view initializer
    printString(s);      // and call the function with the std::string

    printString(static_cast<std::string>(sv)); // okay: we can explicitly cast a std::string_view to a std::string
```

suffix for std::string_veiw literals even though you can obv use c-style literals for initialising std::string_veiw variables.

```
using namespace std::string_literals;      // access the s suffix
using namespace std::string_view_literals; // access the sv suffix

std::cout << "foo\n";   // no suffix is a C-style string literal
std::cout << "goo\n"s;  // s suffix is a std::string literal
std::cout << "moo\n"sv; // sv suffix is a std::string_view literal
```

constexpr std::string_veiw stringname{“this is a string”} is fully supported and is the preffered way for symbolic string constants.

cases where std::string_veiw can’t be used

std::string can’t be used to read a std::string who’s scope has been terminated

```
int main()
{
    std::string_view sv{};

    { // create a nested block
        std::string s{ "Hello, world!" }; // create a std::string local to this nested block
        sv = s; // sv is now viewing s
    } // s is destroyed here, so sv is now viewing an invalid string

    std::cout << sv << '\n'; // undefined behavior

    return 0;
}
```

the scope for the return value of function ends with the function call and since std::string_veiw makes no copies of the initialiser, it can't read the value for future tasks.

```
std::string getName()
{
    std::string s { "Alex" };
    return s;
}

int main()
{
  std::string_view name { getName() }; // name initialized with return value of function
  std::cout << name << '\n'; // undefined behavior

  return 0;
}
```

```
std::string_view getBoolName(bool b)
{
    std::string t { "true" };  // local variable
    std::string f { "false" }; // local variable

    if (b)
        return t;  // return a std::string_view viewing t

    return f; // return a std::string_view viewing f
} // t and f are destroyed at the end of the function

int main()
{
    std::cout << getBoolName(true) << ' ' << getBoolName(false) << '\n'; // undefined behavior

    return 0;
}
```

the variables "t" and "f" in the function will get destroyed at the end of the function and hence the string_veiw reading these std::string variables, will not be able to do so for future reading tasks. this problem only happens because std::string gets destroyed at the end of the function scope unlike c-style literals which stay alive for the whole program

```
std::string_view getBoolName(bool b)
{
    if (b)
        return "true";  // return a std::string_view viewing "true"

    return "false"; // return a std::string_view viewing "false"
} // "true" and "false" are not destroyed at the end of the function

int main()
{
    std::cout << getBoolName(true) << ' ' << getBoolName(false) << '\n'; // ok

    return 0;
}
```

std::string_veiw **can** be used as function parameters though!!

```
std::string_view firstAlphabetical(std::string_view s1, std::string_view s2)
{
    return s1 < s2 ? s1: s2; // uses operator?: (the conditional operator)
}

int main()
{
    std::string a { "World" };
    std::string b { "Hello" };

    std::cout << firstAlphabetical(a, b) << '\n'; // prints "Hello"

    return 0;
}
```

especially here, since the variables “a” & “b” are within the scope of the caller

```
using namespace std::string_literals;
std::string_view name { "Alex"s }; // "Alex"s creates a temporary std::string
std::cout << name << '\n'; // undefined behavior
```

similarly here, the “Alex” literal with std::string suffix will still result in undefined behaviours because “name” is using the std::string literal for initialisation, that’s all, the std::string will go out of scope after that line and leave the std::string_veiw in what we call a **dangling view**.

```
std::string s { "Hello, world!" };
std::string_view sv { s }; // sv is now viewing s

s = "Hello, universe!";    // modifies s, which invalidates sv (s is still valid)
std::cout << sv << '\n';   // undefined behavior
```

modifying the string to be read will also lead to undefined behaviour, as this makes the std::string **invalidated.**

we can revalidate, invalidated objects by setting back to a “good state” or initialise their value to view the string (now modified) again.

```
std::string s { "Hello, world!" };
std::string_view sv { s }; // sv is now viewing s

s = "Hello, universe!";    // modifies s, which invalidates sv (s is still valid)
std::cout << sv << '\n';   // undefined behavior

sv = s;                    // revalidate sv: sv is now viewing s again
std::cout << sv << '\n';   // prints "Hello, universe!"
```

modifying the std::string_view

since std::string_veiw only provides a read-only access to the string, we can’t modify the string but we can change how we want to look on it, strictly for reading purposes.

```cpp
int main()
{
    std::string_view str{ "Peach" };
    std::cout << str << '\n';

    // Remove 1 character from the left side of the view
    str.remove_prefix(1);
    std::cout << str << '\n';

    // Remove 2 characters from the right side of the view
    str.remove_suffix(2);
    std::cout << str << '\n';

    str = "Peach"; // reset the view
    std::cout << str << '\n';

    return 0;
```

however once the modification has taken place, the only method to reset the view is by reassigning the source to the string_veiw.

this is particularly helpful in viewing sub-strings without making any copy where we don't have the need to view the whole string. this also means that std::string_veiw aren't necessarily going to be null terminated unless they are viewing the whole string.

operator

an **operation** is a mathematical process involving zero or more input values (called **operands**) that produces a new value (called an output value). The specific operation to be performed is denoted by a construct (typically a symbol or pair of symbols) called an **operator**.

operator Precedence Table

higher Precedence means Higher Priority over Lower Precedence Operators
L->R indicates how two identical precedence operators are evaluated.

Prec/Ass	Operator	Description	Pattern
1 L->R	::	Global scope (unary)	::name
	::	Namespace scope (binary)	class_name::member_name
2 L->R	()	Parentheses	(expression)
	()	Function call	function_name(arguments)
	type()	Functional cast	type(expression)
	type{}	List init temporary object (C++11)	type{expression}
	[]	Array subscript	pointer[expression]
	.	Member access from object	object.member_name
	->	Member access from object ptr	object_pointer->member_name
	++	Post-increment	lvalue++
	––	Post-decrement	lvalue––
	typeid	Run-time type information	typeid(type) or typeid(expression)
	const_cast	Cast away const	const_cast<type>(expression)
	dynamic_cast	Run-time type-checked cast	dynamic_cast<type>(expression)
	reinterpret_cast	Cast one type to another	reinterpret_cast<type>(expression)
	static_cast	Compile-time type-checked cast	static_cast<type>(expression)
	sizeof...	Get parameter pack size	sizeof...(expression)
	noexcept	Compile-time exception check	noexcept(expression)
	alignof	Get type alignment	alignof(type)
3 R->L	+	Unary plus	+expression
	-	Unary minus	-expression
	++	Pre-increment	++lvalue
	––	Pre-decrement	––lvalue
	!	Logical NOT	!expression
	not	Logical NOT	not expression
	~	Bitwise NOT	~expression
	(type)	C-style cast	(new_type)expression
	sizeof	Size in bytes	sizeof(type) or sizeof(expression)
	co_await	Await asynchronous call	co_await expression (C++20)
	&	Address of	&lvalue
	*	Dereference	*expression
	new	Dynamic memory allocation	new type
	new[]	Dynamic array allocation	new type[expression]
	delete	Dynamic memory deletion	delete pointer
	delete[]	Dynamic array deletion	delete[] pointer

4 L->R	->* .*	Member pointer selector Member object selector	object_pointer->*pointer_to_member object.*pointer_to_member
5 L->R	* / %	Multiplication Division Remainder	expression * expression expression / expression expression % expression
6 L->R	+ -	Addition Subtraction	expression + expression expression - expression
7 L->R	<< >>	Bitwise shift left / Insertion Bitwise shift right / Extraction	expression << expression expression >> expression
8 L->R	<=>	Three-way comparison (C++20)	expression <=> expression
9 L->R	< <= > >=	Comparison less than Comparison less than or equals Comparison greater than Comparison greater than or equals	expression < expression expression <= expression expression > expression expression >= expression
10 L->R	== !=	Equality Inequality	expression == expression expression != expression
11 L->R	&	Bitwise AND	expression & expression
12 L->R	^	Bitwise XOR	expression ^ expression
13 L->R	|	Bitwise OR	expression | expression
14 L->R	&& and	Logical AND Logical AND	expression && expression expression and expression
15 L->R	|| or	Logical OR Logical OR	expression || expression expression or expression
16 R->L	throw co_yield ?: = *= /= %= += -= <<= >>= &= |= ^=	Throw expression Yield expression (C++20) Conditional Assignment Multiplication assignment Division assignment Remainder assignment Addition assignment Subtraction assignment Bitwise shift left assignment Bitwise shift right assignment Bitwise AND assignment Bitwise OR assignment Bitwise XOR assignment	throw expression co_yield expression expression ? expression : expression lvalue = expression lvalue *= expression lvalue /= expression lvalue %= expression lvalue += expression lvalue -= expression lvalue <<= expression lvalue >>= expression lvalue &= expression lvalue |= expression lvalue ^= expression
17 L->R	,	Comma operator	expression, expression

parenthesization

in C++ we can explicitly use parentheses to set the grouping of operands as we desire. This works because parentheses have one of the highest precedence levels, so parentheses generally evaluate before whatever is inside them.

```
x = (y + z + w);   // instead of this
x = y + z + w;     // it's okay to do this

x = ((y || z) && w); // instead of this
x = (y || z) && w;   // it's okay to do this

x = (y *= z); // expressions with multiple assignments still benefit from parenthesis
```

the order of evaluation of operands

the precedence and associativity rules only tell us how operators and operands are grouped and the order in which value computation will occur. They do not tell us the order in which the operands or subexpressions are evaluated.

```
#include <iostream>

int getValue()
{
    std::cout << "Enter an integer: ";

    int x{};
    std::cin >> x;
    return x;
}

void printCalculation(int x, int y, int z)
{
    std::cout << x + (y * z);
}

int main()
{
    printCalculation(getValue(), getValue(), getValue()); // this line is ambiguous

    return 0;
}
```

here the order at which getValue()s will take place is highly ambiguous which means that if we input the values 1 , 2 , 3; we have no way of knowing if x,y,z or z,x,y will get the values in that order.

only way to fix this possible discrepancy is to separately input the function arguments after initially getValuing() their value

```
int main()
{
    int a{ getValue() }; // will execute first
    int b{ getValue() }; // will execute second
    int c{ getValue() }; // will execute third

    printCalculation(a, b, c); // this line is now unambiguous

    return 0;
}
```

binary arithmetic operators

Operator	Symbol	Form	Operation
Addition	+	x + y	x plus y
Subtraction	-	x - y	x minus y
Multiplication	*	x * y	x multiplied by y
Division	/	x / y	x divided by y
Remainder	%	x % y	The remainder of x divided by y

since **/** is integer division devoid of fractional part in answer, unless one of the operands is floating. We can use static_cast to not lose the fractional part.

```
constexpr int x{ 7 };
constexpr int y{ 4 };

std::cout << "int / int = " << x / y << '\n';
std::cout << "double / int = " << static_cast<double>(x) / y << '\n';
std::cout << "int / double = " << x / static_cast<double>(y) << '\n';
std::cout << "double / double = " << static_cast<double>(x) / static_cast<double>(y) << '\n';
```

```
int / int = 1
double / int = 1.75
int / double = 1.75
double / double = 1.75
```

arithmetic assignment operators

Operator	Symbol	Form	Operation
Assignment	=	x = y	Assign value y to x
Addition assignment	+=	x += y	Add y to x
Subtraction assignment	-=	x -= y	Subtract y from x
Multiplication assignment	*=	x *= y	Multiply x by y
Division assignment	/=	x /= y	Divide x by y
Remainder assignment	%=	x %= y	Put the remainder of x / y in x

Prefix & Postfix

Operator	Symbol	Form	Operation
Prefix increment (pre-increment)	++	++x	Increment x, then return x
Prefix decrement (pre-decrement)	--	--x	Decrement x, then return x
Postfix increment (post-increment)	++	x++	Copy x, then increment x, then return the copy
Postfix decrement (post-decrement)	--	x--	Copy x, then decrement x, then return the copy

```
int x { 5 };
int y { ++x }; // x is incremented to 6, x is evaluated to the value 6, and 6 is assigned to y
```

```
int x { 5 };
int y { x++ }; // x is incremented to 6, copy of original x is evaluated to the value 5, and 5 is assigned to y
```

always prefer prefix in normal cases

this could lead to order of evaluation based undefined behaviour and historically led to many bugs

```
int x { 5 };
int value{ add(x, ++x) }; // undefined behavior: is this 5 + 6, or 6 + 6?
// It depends on what order your compiler evaluates the function arguments in
```

there is no **Exponent Operator** in C++ and we have the pow() function in <cmath>

```cpp
#include <cmath>

double x{ std::pow(3.0, 4.0) }; // 3 to the 4th power
```

but the parameters and return value are of type double and this could lead to floating rounding errors even if we pass integers or whole numbers. In most cases integer exponentiation also overflows the integer type and that's why such a function wasn't included in the standard library

```cpp
//We are better off using our own function
#include <cassert> // for assert
#include <cstdint> // for std::int64_t
#include <iostream>

// note: exp must be non-negative
// note: does not perform range/overflow checking, use with caution
constexpr std::int64_t powint(std::int64_t base, int exp)
{
	assert(exp >= 0 && "powint: exp parameter has negative value");

	// Handle 0 case
	if (base == 0)
		return (exp == 0) ? 1 : 0;

	std::int64_t result{ 1 };
	while (exp > 0)
	{
		if (exp & 1)  // if exp is odd
			result *= base;
		exp /= 2;
		base *= base;
	}

	return result;
}

int main()
{
	std::cout << powint(7, 12) << '\n'; // 7 to the 12th power

	return 0;
}
```

as for a safer version which checks for **overflow**

```
#include <cassert> // for assert
#include <cstdint> // for std::int64_t
#include <iostream>
#include <limits> // for std::numeric_limits

// A safer (but slower) version of powint() that checks for overflow
// note: exp must be non-negative
// Returns std::numeric_limits<std::int64_t>::max() if overflow occurs
constexpr std::int64_t powint_safe(std::int64_t base, int exp)
{
    assert(exp >= 0 && "powint_safe: exp parameter has negative value");

    // Handle 0 case
    if (base == 0)
        return (exp == 0) ? 1 : 0;

    std::int64_t result { 1 };

    // To make the range checks easier, we'll ensure base is positive
    // We'll flip the result at the end if needed
    bool negativeResult{ false };

    if (base < 0)
    {
        base = -base;
        negativeResult = (exp & 1);
    }

    while (exp > 0)
    {
        if (exp & 1) // if exp is odd
        {
            // Check if result will overflow when multiplied by base
            if (result > std::numeric_limits<std::int64_t>::max() / base)
            {
                std::cerr << "powint_safe(): result overflowed\n";
                return std::numeric_limits<std::int64_t>::max();
            }

            result *= base;
        }

        exp /= 2;

        // If we're done, get out here
        if (exp <= 0)
            break;

        // The following only needs to execute if we're going to iterate again

        // Check if base will overflow when multiplied by base
        if (base > std::numeric_limits<std::int64_t>::max() / base)
        {
            std::cerr << "powint_safe(): base overflowed\n";
            return std::numeric_limits<std::int64_t>::max();
        }

        base *= base;
    }

    if (negativeResult)
        return -result;

    return result;
}

int main()
{
    std::cout << powint_safe(7, 12) << '\n'; // 7 to the 12th power
    std::cout << powint_safe(70, 12) << '\n'; // 70 to the 12th power (will return the max 64-bit int value)

    return 0;
}
```

comma operator

he comma operator evaluates the left operand, then the right operand, and then returns the result of the right operand.

```
std::cout << (++x, ++y) << '\n'; // increment x and y, evaluates to the right operand
```

```
z = (a, b); // evaluate (a, b) first to get result of b, then assign that value to variable z.
z = a, b; // evaluates as "(z = a), b", so z gets assigned the value of a, and b is evaluated and discarded.
```

even though most time comma operator is used as a separator for parameters inside function definition which would not invoke the comma operator.

Most programmer avoid comma operator except for loops as most cases of use, would be better written as two separate statements'

relational operator

Operator	Symbol	Form	Operation
Greater than	>	x > y	true if x is greater than y, false otherwise
Less than	<	x < y	true if x is less than y, false otherwise
Greater than or equals	>=	x >= y	true if x is greater than or equal to y, false otherwise
Less than or equals	<=	x <= y	true if x is less than or equal to y, false otherwise
Equality	==	x == y	true if x equals y, false otherwise
Inequality	!=	x != y	true if x does not equal y, false otherwise

using floating point numbers with relational operators can be dangerous as they are not precise and can have small rounding errors and is generally **avoided**

one way to bypass this comparison problem is using **epsilon** which is a very small number and is there to diminish the error occurring due to very small numeral differences around 1e-8 from the result we want

```
#include <cmath> // for std::abs()

// absEpsilon is an absolute value
bool approximatelyEqualAbs(double a, double b, double absEpsilon)
{
    // if the distance between a and b is less than or equal to absEpsilon, then a and b are "close enough"
    return std::abs(a - b) <= absEpsilon;
}
```

we can also use relative epsilon as two sets of different numbers could have different scales of "how much difference" can be considered **close enough**.

```
#include <algorithm> // for std::max
#include <cmath>     // for std::abs

// Return true if the difference between a and b is within epsilon percent of the larger of a and b
bool approximatelyEqualRel(double a, double b, double relEpsilon)
{
    return (std::abs(a - b) <= (std::max(std::abs(a), std::abs(b)) * relEpsilon));
}
```

This function may not work for values approaching zero but the workaround for that is complex.

logical operators

Operator	Symbol	Example Usage	Operation
Logical NOT	!	!x	true if x is false, or false if x is true
Logical AND	&&	x && y	true if x and y are both true, false otherwise
Logical OR	||	x || y	true if either (or both) x or y are true, false otherwise

we can also use keywords **and , or , not** instead of symbols

logical NOT

logical NOT has very high precedence and this could lead to some Logical Errors.

```cpp
#include <iostream>

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

    if (!x > y)
        std::cout << x << " is not greater than " << y << '\n';
    else
        std::cout << x << " is greater than " << y << '\n';

    return 0;
}
```

will return 5 is greater than 7 as the expression `!x > y` actually evaluates as `(!x) > y`. Since *x* is 5, !x evaluates to *0*, and `0 > y` is false, so the *else* statement executes! and we will need to use `(!x > y)` to properly execute the code.

logical OR

New programmers sometimes try this:

```cpp
if (value == 0 || 1) // incorrect: if value is 0, or if 1
```

When `1` is evaluated, it will implicitly convert to `bool` `true`. Thus this conditional will always evaluate to `true`.

If you want to compare a variable against multiple values, you need to compare the variable multiple times:

```cpp
if (value == 0 || value == 1) // correct: if value is 0, or if value is 1
```

logical AND

in order for Logical AND to return true, all expressions must evaluate to true and if even one (leftist) returns false, the operator will return false immediately. This could lead to **side effects** which are unexpected.

```cpp
if (x == 1 && ++y == 2)
    // do something
```

If x is not equal to 1, then y will never be prefixed.

Logical AND has higher precedence over Logical OR but its better to parenthesize.

de Morgan's Laws

`!(x && y)` is equivalent to `!x || !y`
`!(x || y)` is equivalent to `!x && !y`

There is **no Logical XOR** in C++, but != will similarly work for bool operands (ex **if(a != b != c) ...**)