# Notes from cplusplus.com

14/03/2021

## Basics

* Object oriented programming
* Class is equivalent to structure
* Generic programming
* Vectors (array, arrayLists): Containers of similar data types
* C++ vs C:
* Almost everything in C is there in C++.
* C++ give flexibilities in terms of performance.
* If you want to optimize a program, you will use C++. Extremely computationally heavy.

#include <iostream> // Preprocessor directive

int main() { //You always have the main function. Irrespective of its position in the code, it is called at the beginning when program is run

    std::cout << "Hello world\n"; //std standard: namespace, iostream is a namespace which has cout

    return 0;

}

* Namespaces are used for avoiding conflicts for functions of same name.
  + - Ostream (class) 🡪 cout (object)
    - Cout stands for character output device
* << : operator takes the right hand data and give it as an input to the right hand side object
* Hardcoding is manually typing in values inside our code
* Preprocessor directives (those that begin by #) are out of this general rule since they are not statements. They are lines read and processed by the preprocessor before proper compilation begins. Preprocessor directives must be specified in their own line and, because they are not statements, do not have to end with a semicolon (;).

#include <iostream> //reprocessor

//using directive

// using namespace std;

using std::cout;

using std::cin;

// namespace std;

int main() { //You alwys have the main function

    int slices = 5; //declaration and initialization

    slices = 5; //Assignment is more like storing things

    cout << "Yo man, how many slices of pizza you want? ";

    std::cin >> slices;

    std::cout << "Hello world\n" << slices << "\nslices of pizza";

    //std standard: namespace,  iostream is a namespace which has cout

    return 0;

}

Style guide for c++: How to properly write a program.

C++ is case sensitive.

## Comments

// line comment

/\* block comment \*/

Accessing namespace can be done by two ways

1. an *unqualified* manner (without the std:: prefix). Using statement

using namespace std;

1. *qualified manner* (std::cout), while the second qualifies it directly within the *namespace* std (as std::cout).

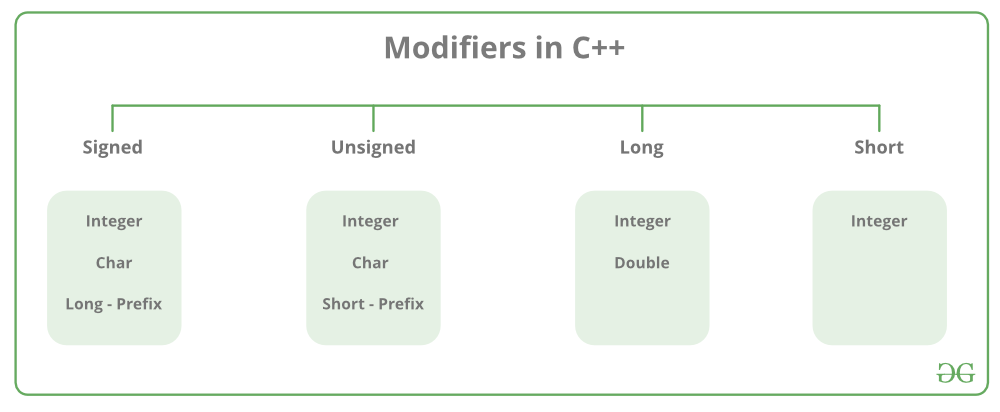
## What is an identifier?

A valid identifier is a sequence of one or more letters, digits, or underscore characters (\_). Spaces, punctuation marks, and symbols cannot be part of an identifier

## Data types and variables

* **Character types:** They can represent a single character, such as 'A' or '$'. The most basic type is char, which is a one-byte character. Other types are also provided for wider characters.
* **Numerical integer types:** They can store a whole number value, such as 7 or 1024. They exist in a variety of sizes, and can either be *signed* or *unsigned*, depending on whether they support negative values or not.
* **Floating-point types:** They can represent real values, such as 3.14 or 0.01, with different levels of precision, depending on which of the three floating-point types is used.
* **Boolean type:** The boolean type, known in C++ as bool, can only represent one of two states, true or false.

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



Detailed table

|  |  |  |
| --- | --- | --- |
| **Group** | **Type names** | **Notes on size / precision** |
| Character types | **char** | Exactly one byte in size. At least 8 bits. |
| **char16\_t** | Not smaller than char. At least 16 bits. |
| **char32\_t** | Not smaller than char16\_t. At least 32 bits. |
| **wchar\_t** | Can represent the largest supported character set. |
| Integer types (signed) | **signed char** | Same size as char. At least 8 bits. |
| *signed* **short** *int* | Not smaller than char. At least 16 bits. |
| *signed* **int** | Not smaller than short. At least 16 bits. |
| *signed* **long** *int* | Not smaller than int. At least 32 bits. |
| *signed* **long long** *int* | Not smaller than long. At least 64 bits. |
| Integer types (unsigned) | **unsigned char** | (same size as their signed counterparts) |
| **unsigned short** *int* |
| **unsigned** *int* |
| **unsigned long** *int* |
| **unsigned long long** *int* |
| Floating-point types | **float** |  |
| **double** | Precision not less than float |
| **long double** | Precision not less than double |
| Boolean type | **bool** | 1 bit |
| Void type | **void** | no storage |
| Null pointer | **decltype(nullptr)** |  |

|  |  |  |
| --- | --- | --- |
| **Data Type** | **Size (in bytes)** | **Range** |
| short int | 2 | -32,768 to 32,767 |
| unsigned short int | 2 | 0 to 65,535 |
| unsigned int | 4 | 0 to 4,294,967,295 |
| int | 4 | -2,147,483,648 to 2,147,483,647 |
| long int | 8 | -2,147,483,648 to 2,147,483,647 |
| unsigned long int | 8 | 0 to 4,294,967,295 |
| long long int | 8 | -(2^63) to (2^63)-1 |
| unsigned long long int | 8 | 0 to 18,446,744,073,709,551,615 |
| signed char | 1 | -128 to 127 |
| unsigned char | 1 | 0 to 255 |
| float | 4 |  |
| double | 8 |  |
| long double | 12 |  |
| wchar\_t | 2 or 4 | 1 wide character |

***Note:*** *in the panel above that other than char (which has a size of exactly one byte), none of the fundamental types has a standard size specified (but a minimum size, at most). Therefore, the type is not required (and in many cases is not) exactly this minimum size. This does not mean that these types are of an undetermined size, but that there is no standard size across all compilers and machines; each compiler implementation may specify the sizes for these types that fit the best the architecture where the program is going to run.*

***Note****: We cannot declare void type*

#include <iostream>

int main(){

    wchar\_t number = L'ם';

    //wchar\_t is used to represent characters that require more memory to represent them than a single char.

    float area = 24.56;

    //For floating-point types, the size affects their precision, by having more or less bits for their significant and exponent.

    bool isDone = true;

    int myNUmber = 10;

    int a, b, c;

    char test = 'A';

    //void, which identifies the lack of type

    //type nullptr, which is a special type of pointer

    std::cout << "This is my number: " << number << std::endl;

}

### Initializing a variable

Three types

1. C-like initialization

type identifier = initial\_value;

1. Constructor initialization

type identifier (initial\_value);

1. Uniform initialization

type identifier {initial\_value};

    int num1 = 0; //c-like

    int num2(20); //constructor

    int num3{121}; //uniform

### Type deduction and decltype

    int num4 = 0;

    auto num5 = num4;

    decltype(num4) num6;

## Constants

These are four valid numbers with decimals expressed in C++. The first number is PI, the second one is the number of Avogadro, the third is the electric charge of an electron (an extremely small number) -all of them approximated-, and the last one is the number *three* expressed as a floating-point numeric literal.

Character and string literals are enclosed in quotes:

|  |  |
| --- | --- |
| 1 2 3 4 | 'z'  'p'  "Hello world"  "How do you do?" |

Escape sequence

|  |  |
| --- | --- |
| **Escape code** | **Description** |
| \n | newline |
| \r | carriage return |
| \t | tab |
| \v | vertical tab |
| \b | backspace |
| \f | form feed (page feed) |
| \a | alert (beep) |
| \' | single quote (') |
| \" | double quote (") |
| \? | question mark (?) |
| \\ | backslash (\) |

Character and string prefixes

|  |  |
| --- | --- |
| **Prefix** | **Character type** |
| u | char16\_t |
| U | char32\_t |
| L | wchar\_t |
| u8 | The string literal is encoded in the executable using UTF-8 |
| R | The string literal is a raw string |

## Preprocessor definition (#define)

Another mechanism to name constant values is the use of preprocessor definitions. They have the following form:  
  
#define identifier replacement

#define PI 3.14159

#define NEWLINE '\n'

We can even replace default values using preprocessor definition

## Operators

Operators perform operations on data types

### Assignment operator (=)

Assigns contents of right hand side contents to left hand side variable

#include <iostream>

int main(){

    int a,b;

    a = 5;

    b = 13;

    a = b; // Assign contents at memory location of b to a.

    b = 20; // b changes but a doesn't because they are different

    // variables pointing to two different locations

    int x = 10;

    int y = 2 + (x=5); // Assignment operations can be evaluated inside other expressions

    // Above operation is equivalent to 1. x = 5 and y = 2 + x; (Assignment operations precedes)

    // x becomes 5 and y = 7

    std::cout << "value of x is: " << x << " and value of y is: " << y << std::endl;

}

Output:

value of x is: 5 and value of y is: 7

### Arithmetic operators (+, -, \*, /, %)

|  |  |
| --- | --- |
| **operator** | **description** |
| + | addition |
| - | subtraction |
| \* | multiplication |
| / | division |
| % | modulo |

Modulo operators gives remainder of the integral division.

### Compound assignment (+=, -=, \*=, /=, %=, >>=, <<=, &=, ^=, |=)

Modify and assign at the same time by this operator

|  |  |
| --- | --- |
| **expression** | **equivalent to...** |
| y += x; | y = y + x; |
| x -= 5; | x = x - 5; |
| x /= y; | x = x / y; |
| price \*= units + 1; | price = price \* (units+1); |

### Increment and decrement operator (++, --)

Increase value by one

|  |  |
| --- | --- |
| ++x;  x+=1;  x=x+1; |  |

are all equivalent in its functionality; the three of them increase by one the value of x.

In the case that the increase operator is used as a prefix (++x ~ x = x + 1) of the value, the expression evaluates to the final value of x, once it is already increased. On the other hand, in case that it is used as a suffix (x++ ~ x + 1), the value is also increased, but the expression evaluates to the value that x had before being increased.

     //Increment and decrement operator

     int z; x = 10;

     z = x++; // z = 11 and x = 10 i.e. no change in x

     std::cout << "Result of z = x++" <<std::endl;

     std::cout << "value of z: " << z <<std::endl;

     std::cout << "value of x: " << x <<std::endl;

     z = ++x;  // z = 12 ans x = 11 i.e. increament in x

     std::cout << "Result of z = ++x" <<std::endl;

     std::cout << "value of z: " << z <<std::endl;

     std::cout << "value of x: " << x <<std::endl;

### Relational operator ( ==, !=, >, <, >=, <= )

Evaluates to true or false

|  |  |
| --- | --- |
| **operator** | **description** |
| == | Equal to |
| != | Not equal to |
| < | Less than |
| > | Greater than |
| <= | Less than or equal to |
| >= | Greater than or equal to |

    //Relational operators (==, !=, <=, >=, >, <)

    int num1, num2; num1 = 10; num2 = 20;

    std::cout << "Values of num1 and num2 are: " << num1 << " and " << num2 <<std::endl;

    std::cout << "Result of (num1 == num2): " << (num1 == num2) <<std::endl;

    std::cout << "Result of (num1 != num2): " << (num1 != num2) <<std::endl;

    std::cout << "Result of (num1 >= num2): " << (num1 >= num2) <<std::endl;

    std::cout << "Result of (num1 <= num2): " << (num1 <= num2) <<std::endl;

    std::cout << "Result of ((num1 = 21) <= num2): " << ((num1 = 21) <= num2) <<std::endl;

    // first assignment  num1 = 21 occurs, then expression is evaluated

### Logical operators ( !, &&, || )

|  |  |  |
| --- | --- | --- |
| **&& OPERATOR (and)** | | |
| **a** | **b** | **a && b** |
| true | true | true |
| true | false | false |
| false | true | false |
| false | false | false |

|  |  |  |
| --- | --- | --- |
| **|| OPERATOR (or)** | | |
| **a** | **b** | **a || b** |
| true | true | true |
| true | false | true |
| false | true | true |
| false | false | false |

These are used when more than one relational operation are required to be evaluated.

***Note:*** *When using the logical operators, C++ only evaluates what is necessary from left to right to come up with the combined relational result, ignoring the rest*

|  |  |
| --- | --- |
| *operator* | *short-circuit* |
| *&&* | *if the left-hand side expression is false, the combined result is false (the right-hand side expression is never evaluated).* |
| *||* | *if the left-hand side expression is true, the combined result is true (the right-hand side expression is never evaluated).* |

    //Logical operators (!, &&, ||)

    num1 = 20; num2 = 30;

    std::cout << "Values of num1 and num2 are: " << num1 << " and " << num2 <<std::endl;

    std::cout << "Result of (num1 <= num2) || (++num1 != 20): " << ((num1 <= num2) || (++num1 != 20)) <<std::endl;

    std::cout << "Values of num1 and num2 are: " << num1 << " and " << num2 <<std::endl;

    //for OR (||), if the left-hand side expression is true, the combined result is true (the right-hand side expression is never evaluated).

    std::cout << "Result of (num1 <= num2) || (++num1 != 20): " << ((num1 > num2) && (++num1 != 22)) <<std::endl;

    std::cout << "Values of num1 and num2 are: " << num1 << " and " << num2 <<std::endl;

    //For AND (&&), if the left-hand side expression is false, the combined result is false (the right-hand side expression is never evaluated).

### Conditional ternary operator (?)

The conditional operator evaluates an expression, returning one value if that expression evaluates to true, and a different one if the expression evaluates as false. Its syntax is:  
  
condition ? result1 : result2

If condition is true, the entire expression evaluates to result1, and otherwise to result2.

    //Conditional ternary operator

    num1 = (7 == 5) ? 5 : 4;

    std::cout << "Result of num1 = (7 == 5) ? 5 : 4 is " << num1 << std::endl;

    num1 = (7 == 5 + 2) ? 5 : 4;

    std::cout << "Result of num1 = (7 == 5 + 2) ? 5 : 4 is " << num1 << std::endl;

### Comma Operator (,)

When one or more operations are required to be performed and end result is obtained with rightmost expression. From left to right expressions are evaluated

    // Comma operator

    num1 = (num1 = 10, num2 = 20, num1 + num2); // should assign num1 to 30

    std::cout << "Result of num1 = (num1 = 10, num2 = 20, num1 + num2) is " << num1 << std::endl;

### Bitwise operators ( &, |, ^, ~, <<, >> )

|  |  |  |
| --- | --- | --- |
| **operator** | **asm equivalent** | **description** |
| & | AND | Bitwise AND |
| | | OR | Bitwise inclusive OR |
| ^ | XOR | Bitwise exclusive OR |
| ~ | NOT | Unary complement (bit inversion) |
| << | SHL | Shift bits left |
| >> | SHR | Shift bits right |

### Explicit type casting operator

Type casting operators allow converting a value of a given type to another type. There are several ways to do this in C++. The simplest one, which has been inherited from the C language, is to precede the expression to be converted by the new type enclosed between parentheses.

    //explicit type casting

    float num5 = 3.142;

    int myInt;

    myInt = (int) num5;

    std::cout << "Value of casted myInt from num5 is " << myInt << std::endl;

    // Everything after decimal place is lost

### Precedence of operators

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Level** | **Precedence group** | **Operator** | **Description** | **Grouping** |
| 1 | Scope | :: | scope qualifier | Left-to-right |
| 2 | Postfix (unary) | ++ -- | postfix increment / decrement | Left-to-right |
| () | functional forms |
| [] | subscript |
| . -> | member access |
| 3 | Prefix (unary) | ++ -- | prefix increment / decrement | Right-to-left |
| ~ ! | bitwise NOT / logical NOT |
| + - | unary prefix |
| & \* | reference / dereference |
| new delete | allocation / deallocation |
| sizeof | parameter pack |
| (*type*) | C-style type-casting |
| 4 | Pointer-to-member | .\* ->\* | access pointer | Left-to-right |
| 5 | Arithmetic: scaling | \* / % | multiply, divide, modulo | Left-to-right |
| 6 | Arithmetic: addition | + - | addition, subtraction | Left-to-right |
| 7 | Bitwise shift | << >> | shift left, shift right | Left-to-right |
| 8 | Relational | < > <= >= | comparison operators | Left-to-right |
| 9 | Equality | == != | equality / inequality | Left-to-right |
| 10 | And | & | bitwise AND | Left-to-right |
| 11 | Exclusive or | ^ | bitwise XOR | Left-to-right |
| 12 | Inclusive or | | | bitwise OR | Left-to-right |
| 13 | Conjunction | && | logical AND | Left-to-right |
| 14 | Disjunction | || | logical OR | Left-to-right |
| 15 | Assignment-level expressions | = \*= /= %= += -= >>= <<= &= ^= |= | assignment / compound assignment | Right-to-left |
| ?: | conditional operator |
| 16 | Sequencing | , | comma separator | Left-to-right |

When an expression has two operators with the same precedence level, *grouping* determines which one is evaluated first: either left-to-right or right-to-left.

15/03/2021

## Basic Input/Output

|  |  |
| --- | --- |
| **stream** | **description** |
| cin | standard input stream |
| cout | standard output stream |
| cerr | standard error (output) stream |
| clog | standard logging (output) stream |

C++ uses a convenient abstraction called *streams* to perform input and output operations in sequential media such as the screen, the keyboard or a file.

### Standard output (cout)

On most program environments, the standard output by default is the screen, and the C++ stream object defined to access it is cout.

The << operator inserts the data that follows it into the stream that precedes it.

***Note****: The endl manipulator produces a newline character, exactly as the insertion of '\n' does; but it also has an additional behavior: the stream's buffer (if any) is flushed, which means that the output is requested to be physically written to the device, if it wasn't already. This affects mainly fully buffered streams, and cout is (generally) not a fully buffered stream. Still, it is generally a good idea to use endl only when flushing the stream would be a feature and '\n' when it would not. Bear in mind that a flushing operation incurs a certain overhead, and on some devices it may produce a delay.*

*Q: What is stream buffer, and flushing the buffer?*

#include <iostream>

int main(){

    std::cout << "this is one sentence." << " This is another sentence." << std::endl;

    std::cout << "We can use escape sequence here.\n" << "It doesn't neccessarily flush the stream's buffer." << std::endl;

    std::cout << "Flushing stream buffer is done with endl object. But it has a overhead." << std::endl;

}

### Standard Input

In most program environments, the standard input by default is the keyboard, and the C++ stream object defined to access it is cin.

The extraction operation on cin uses the type of the variable after the >> operator to determine how it interprets the characters read from the input; if it is an integer, the format expected is a series of digits, if a string a sequence of characters, etc.

***Note****: if the user enters something else that cannot be interpreted as an integer? In this case, the extraction operation fails. And this, by default, lets the program continue without setting a value for variable i, producing undetermined results.*

***Note:*** *Any kind of space is used to separate two consecutive input operations; this may either be a space, a tab, or a new-line character.*

    int num1;

    std::cin >> num1;

    float num2;

    std::cin >> num2;

    std::cout << "This is my number: " << num1 << "\nThis is another number: " << num2 <<std::endl;

    std::cin >> num1 >> num2;

    std::cout << "This is my number: " << num1 << "\nThis is another number: " << num2 <<std::endl;

### Cin and Strings

In case of strings, extraction of no more than a single word is possible as, space, tab and new line means termination of the stream.

There is a function called getline, it is used to an entire line and is terminated by newline character (ENTER on keyboard)

Syntax:

getline(stream, stringSequenceToBePrinted);

    //To get a line as input stream

    std::string myStr;

    std::cout  << "What is your name? ";

    getline (std::cin, myStr); //If I have used cin before this, the stream buffer is flushed, then it will not take input from user

    std::cout << "Hello " << myStr << "!" << std::endl;

    std::cout << "What is your favourite sport? ";

    getline (std::cin, myStr);

    std::cout << "I like " << myStr;

### String stream

    //String Stream by adding header <sstream>

    float price; int quantity;

    std::cout << "Enter the price: ";

    getline(std::cin, myStr);

    std::stringstream(myStr) >> price;

    std::cout << "Enter the quantity: ";

    getline(std::cin, myStr);

    std::stringstream(myStr) >> quantity;

    std::cout << "The total price is "<< price \* quantity;

16/03/2021

## Statements and flow control

A program may repeat segments of code, or take decisions and bifurcate.

Flow control statements require a generic statement as a part of its syntax. It can be a compound statement or a single statement

{state1, state2, state3} this is a compound statement

### Selection statement

if (condition) statement

Selection statements with if can also specify what happens when the condition is not fulfilled, by using the else keyword to introduce an alternative statement

if (condition) statement1 else statement2

#include <iostream>

#include <string>

#include <sstream>

int main(){

    std::string mystr;

    int num1;

    std::cout << "Enter a number: ";

    getline(std::cin, mystr);

    std::stringstream(mystr) >> num1;

    if (num1 > 0) std::cout << "You have entered: "<< num1 << ". It is a positive number.";

    else if (num1 < 0) std::cout << "You have entered: "<< num1 << ". It is a negative number.";

    else std::cout << "You have entered zero.";

}

### Iteration statements (loops)

Loops repeat a statement a certain number of times, or while a condition is fulfilled. They are introduced by the keywords while, do, and for.

#### The while loop

while (expression) statement

A thing to consider with while-loops is that the loop should end at some point, and thus the statement shall alter values checked in the condition in some way, so as to force it to become false at some point. Otherwise, the loop will continue looping forever.

    //While loop

    int num2 = 200;

    while (num2 > 0) {

        std::cout << num2 <<", ";

        num2--;

    }

    std::cout << " 0.";

#### The do-while loop

do statement while (condition);  
  
It behaves like a while-loop, except that condition is evaluated after the execution of statement instead of before, guaranteeing at least one execution of statement, even if condition is never fulfilled.

***Note****: The do-while loop is usually preferred over a while-loop when the statement needs to be executed at least once, such as when the condition that is checked to end of the loop is determined within the loop statement itself.*

    //Do-while loop.

    do {

        num2++;

        getline (std::cin, mystr);

        std::cout << "You entered: " << mystr << std::endl;

        std::cout << "num2: " << num2 << std::endl;

    } while ((mystr != "Exit") && (num2 < 10));

#### The for loop

for (initialization; condition; increase) statement;  
  
Like the while-loop, this loop repeats statement while condition is true. But, in addition, the for loop provides specific locations to contain an initialization and an increase expression, executed before the loop begins the first time, and after each iteration, respectively.

1. initialization is executed. Generally, this declares a counter variable, and sets it to some initial value. This is executed a single time, at the beginning of the loop.
2. condition is checked. If it is true, the loop continues; otherwise, the loop ends, and statement is skipped, going directly to step 5.
3. statement is executed. As usual, it can be either a single statement or a block enclosed in curly braces { }.
4. increase is executed, and the loop gets back to step 2.
5. the loop ends: execution continues by the next statement after it.

***Note:*** *Because each of the fields is executed in a particular time in the life cycle of a loop, it may be useful to execute more than a single expression as any of initialization, condition, or statement. Unfortunately, these are not statements, but rather, simple expressions, and thus cannot be replaced by a block. As expressions, they can, however, make use of the comma operator (,): This operator is an expression separator, and can separate multiple expressions where only one is generally expected.*

https://www.cplusplus.com/doc/tutorial/control/for_loop.png

    //For loop

    for (;(num2 < num1);num2++){

        std::cout << "Num1 = " << num1 << "\tNum2 = " << num2 <<std::endl;

    }

    //For loop with comma operator

    for(int i=1, j = 1; (i - j < 200); i++, j--)

    {std::cout << "i = " << i << "\tj = " << j <<std::endl;}

#### Range-based for loop

for ( declaration : range ) statement;

 Ranges are sequences of elements, including arrays, containers, and any other type supporting the functions begin and end; e.g. strings are sequences of characters.

This loop is automatic and does not require the explicit declaration of any counter variable.

Range based loops usually also make use of type deduction for the type of the elements with auto.

Q: Does range based loop has a provision for increment?

#include <iostream>

#include <string>

int main(){

    std::string myStr;

    getline(std::cin, myStr);

    for(char c:myStr){

        std::cout << "\'" << c << "\' \_ ";

    }

}

We can also use auto c:myStr here.

### Jump statements

#### The break statement

break leaves a loop, even if the condition for its end is not fulfilled.

#### The continue statement

The continue statement causes the program to skip the rest of the loop in the current iteration, as if the end of the statement block had been reached, causing it to jump to the start of the following iteration.

#### The goto statement

goto allows to make an absolute jump to another point in the program. This unconditional jump ignores nesting levels, and does not cause any automatic stack unwinding.

// goto loop example

#include <iostream>

using namespace std;

int main ()

{

  int n=10;

mylabel:

  cout << n << ", ";

  n--;

  if (n>0) goto mylabel;

  cout << "liftoff!\n";

}

### Switch statement

Its purpose is to check for a value among a number of possible constant expressions. It is something similar to concatenating if-else statements, but limited to constant expressions.

switch (expression)

{

  case constant1:

     group-of-statements-1;

     break;

  case constant2:

     group-of-statements-2;

     break;

  .

  .

  .

  default:

     default-group-of-statements

}

It works in the following way: switch evaluates expression and checks if it is equivalent to constant1; if it is, it executes group-of-statements-1 until it finds the break statement. When it finds this break statement, the program jumps to the end of the entire switch statement (the closing brace).

Finally, if the value of expression did not match any of the previously specified constants (there may be any number of these), the program executes the statements included after the default: label, if it exists (since it is optional).

***Note****: break statements are needed after each group of statements for a particular label. If break is not included, all statements following the case (including those under any other labels) are also executed, until the end of the switch block or a jump statement. It can also be useful to execute the same group of statements for different possible values.*

## Functions

//Function declaration:  similar to reservation of memory for variables

dataType functionName(datatype1 param1, datatype2 param2);

//Function definition

datatype functionName(param1, param2…){

//function body

}

//Call to this function

functionName(arg1, arg2);

The function is called from within main, the control is passed to function being called: here, execution of main is stopped, and will only resume once the function being called ends. At the moment of the function call, the value of both arguments (5 and 3) are copied to the local variables int a and int b within the function.

***Note:*** *Each parameter looks very much like a regular variable declaration (for example: int x), and in fact acts within the function as a regular variable which is local to the function. The purpose of parameters is to allow passing arguments to the function from the location where it is called from.*

***Note****: The arguments passed to subtraction are variables instead of literals. That is also valid, and works fine. each function call is itself an expression that is evaluated as the value it returns*

### Void functions:

Don’t return any value but does something. Like print function.

***Note****: void can also be used in the function's parameter list to explicitly specify that the function takes no actual parameters when called.*

In C++, an empty parameter list can be used instead of void with same meaning.

### The return value of main

If the execution of main ends normally without encountering a return statement the compiler assumes the function ends with an implicit return statement return 0;

|  |  |
| --- | --- |
| **value** | **description** |
| 0 | The program was successful |
| [EXIT\_SUCCESS](https://www.cplusplus.com/EXIT_SUCCESS) | The program was successful (same as above). This value is defined in header [<cstdlib>](https://www.cplusplus.com/%3Ccstdlib%3E). |
| [EXIT\_FAILURE](https://www.cplusplus.com/EXIT_FAILURE) | The program failed. This value is defined in header [<cstdlib>](https://www.cplusplus.com/%3Ccstdlib%3E). |

### Arguments passed by value and by reference

Pass by value: This means that, when calling a function, what is passed to the function are the values of these arguments on the moment of the call, which are copied into the variables represented by the function parameters.

Pass by reference: Values of actual variables are modified. To gain access to its arguments, the function declares its parameters as *references*. In C++, references are indicated with an ampersand (&) following the parameter type.

***Note****: When a variable is passed by reference, what is passed is no longer a copy, but the variable itself, the variable identified by the function parameter, becomes somehow associated with the argument passed to the function, and any modification on their corresponding local variables within the function are reflected in the variables passed as arguments in the call.*

### Efficiency considerations and const references

Arguments by reference do not require a copy. The function operates directly on (aliases of) the strings passed as arguments, and, at most, it might mean the transfer of certain pointers to the function.   
  
On the flip side, functions with reference parameters are generally perceived as functions that modify the arguments passed, because that is why reference parameters are actually for.  
  
The solution is for the function to guarantee that its reference parameters are not going to be modified by this function.

Efficiency depends on whether the function params are passed by value or passed by reference.

Making copies of variable of compound type which are large in sizes has some overhead. Passing by reference does not involve copying of variables to params and can actually be treated as aliases to variables and has no overhead.

***Note****: The solution is for the function to guarantee that its reference parameters are not going to be modified by this function. This can be done by qualifying the parameters as constant.*

string concatenate (const string& a, const string& b)

{

  return a+b;

}

const references provide functionality similar to passing arguments by value, but with an increased efficiency for parameters of large types.

### Inline Functions

Calling a function generally causes a certain overhead (stacking arguments, jumps, etc...), and thus for very short functions, it may be more efficient to simply insert the code of the function where it is called, instead of performing the process of formally calling a function.

inline specifier informs the compiler that inline expansion is preferred over the usual function call mechanism.

***Note:*** *In C++, optimization is a task delegated to the compiler, which is free to generate any code for as long as the resulting behavior is the one specified by the code.*

### Default values of parameters

Functions can also have optional parameters, for which no arguments are required in the call.

For this, the function shall include a default value for its last parameter, which is used by the function when called with fewer arguments.

#include <iostream>

#include <string>

int add (int a, int b = 1){

    return a+b;

}

int main(){

    std::cout << "The sum 523 + 211 = " << add(523, 211) << std::endl;

    std::cout << "The result of add(523) = " << add(523) << std::endl;

}

### Declaring functions

Functions cannot be called before they are declared. If main were defined before the other functions, this would break the rule that functions shall be declared before being used, and thus would not compile.

The prototype of a function can be declared without actually defining the function completely, giving just enough details to allow the types involved in a function call to be known.

The prototype of a function can be declared without actually defining the function completely, giving just enough details to allow the types involved in a function call to be known.

//function declarations

int substract (int, int);

int multiply (int, int );

int divide (int a, int b);

int remainder (int, int);

### Recursivity

Recursivity is the property that functions have to be called by themselves. It is useful for some tasks, such as sorting elements, or calculating the factorial of numbers.

#include <iostream>

int function1(int);

int function2(int);

bool validateFunction1(int a, int sum){

    return (sum == (a\*(a+1)/2));

}

int main(){

    int myInt;

    std::cout << "Enter a number: ";

    std::cin >> myInt;

    int sum = function1(myInt);

    std::cout << "The sum of integers upto " << myInt << " is " << sum << std::endl;

    if (validateFunction1(myInt, sum)) std::cout << "The results are correct.";

    else std::cout << "Recursive function failed.";

}

int function1(int a){

    if (a>0){

        return (a+function1(a-1));

    } else return 0;

}

//sum of odd numbers up to a given number

int function2(int a){

    if (a>1){

        a--;

        std::cout << a << std::endl;

        return (a+function2(a-1));

    } else return 0;

}

### Overloards and templates (Function overloading)

### Overloaded functions

In C++, two different functions can have the same name if their parameters are different; either because they have a different number of parameters, or because any of their parameters are of a different type.

The compiler knows which one to call in each case by examining the types passed as arguments when the function is called.

***Note****: A function cannot be overloaded only by its return type. At least one of its parameters must have a different type.*

### Function template

Overloaded functions have same body but different parameters types.

C++ has the ability to **define functions with generic types**, known as *function templates*.

 Function template is preceded by the template keyword and a series of template parameters enclosed in angle-brackets <>.

template <template-parameters> function-declaration

 Template parameters can be generic template types by specifying either the class or typename keyword followed by an identifier.

e.g

template <class T>

T sum (T a, T b)

{

  return a+b;

}

template <typename SomeType>  // typename ~ class (both keywords means the same thing)

SomeType sum (SomeType a, SomeType b)

{

  return a+b;

}

It can be used as the type for parameters, as return type, or to declare new variables of this type. In all cases, it represents a generic type that will be determined on the moment the template is instantiated.

Instantiating a template is applying the template to create a function using particular types or values for its template parameters.

A template is instantiated as following

name <template-arguments> (function-arguments)

int main() {

    std::cout << sum3<int>(20.01, 10) << std::endl;

    std::cout << sum3<float>(20.3, 12.25) << std::endl;

}

The compiler is even able to deduce the data type automatically without having to explicitly specify it within angle brackets. Therefore, instead of explicitly specifying the template arguments with

***Note****: numerical literals are always of a specific type: Unless otherwise specified with a suffix, integer literals always produce values of type int, and floating-point literals always produce values of type double.*

### Non-type template arguments

The template parameters can not only include types introduced by class or typename, but can also include expressions of a particular type.

template <class Type1, int myInt> //expression

Type1 constantAdd (Type1 a)

{

  return a+myInt;

}

    std::cout << constantAdd<int, 3>(20) << std::endl;

    std::cout << constantAdd<float, 1000>(10.25) << std::endl;

***Note****: the value of template parameters is determined on compile-time to generate a different instantiation of the function constantAdd, and thus the value of that argument is never passed during runtime.*

***Note****: Template parameters can never be passed a variable, as they are different instantiation of same function.*

## Name visibility

### Scopes

An entity declared outside any block has *global scope*, meaning that its name is valid anywhere in the code. While an entity declared within a block, such as a function or a selective statement, has *block scope*, and is only visible within the specific block in which it is declared, but not outside it.

The visibility of an entity with *block scope* extends until the end of the block, including inner blocks. Nevertheless, an inner block, because it is a different block, can re-utilize a name existing in an outer scope to refer to a different entity; in this case, the name will refer to a different entity only within the inner block, hiding the entity it names outside. While outside it, it will still refer to the original entity.

***Note****: Variables declared in declarations that introduce a block, such as function parameters and variables declared in loops and conditions (such as those declared on a for or an if) are local to the block they introduce.*

### Namespace

Namespaces allow us to group named entities that otherwise would have *global scope* into narrower scopes, giving them *namespace scope*. This allows organizing the elements of programs into different logical scopes referred to by names.

namespace identifier

{

  named\_entities

}

//Accessing variables from a namespace

myNamespace::a

myNamespace::b

***Note****: Namespaces can be split: Two segments of a code can be declared in the same namespace*

namespace foo { int a; }

namespace bar { int b; }

namespace foo { int c; }

 It would be possible to first use the objects of one namespace and then those of another one by splitting the code in different blocks.

Existing namespaces can be aliased with new names, with the following syntax:  
  
namespace new\_name = current\_name;

### Storage classes

The storage for variables with *global* or *namespace scope* is allocated for the entire duration of the program. This is known as *static storage*, and it contrasts with the storage for *local variables* (those declared within a block). These use what is known as automatic storage.

But there is another substantial difference between variables with *static storage* and variables with *automatic storage*:  
- Variables with *static storage* (such as global variables) that are not explicitly initialized are automatically initialized to zeroes.  
- Variables with *automatic storage* (such as local variables) that are not explicitly initialized are left uninitialized, and thus have an undetermined value.

20/03/2021

## Arrays

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

Syntax:

type name [elements];

name: identifier, elements = number of array elements and type = any valid type

    int myArr1[5] = {1, 5, 3, 5,9}; //Initializing with value.

    int myArr2[5] = {1,2,3}; //Will make last two elements zero.

    int myArr3[] = {10, 5, 6}; //Compliler will take size = 3.

    int myArr4 [] {10, 20, 5}; //c++ 11 more generic initialization

***NOTE****: The elements field within square brackets [], representing the number of elements in the array, must be a constant expression, since arrays are blocks of static memory whose size must be determined at compile time.*

If declared with less, the remaining elements are set to their default values (which for fundamental types, means they are filled with zeroes)

Static arrays, and those declared directly in a namespace (outside any function), are always initialized. If no explicit initializer is specified, all the elements are default-initialized (with zeroes, for fundamental types)

***Note****: In C++, it is syntactically correct to exceed the valid range of indices for an array. This can create problems, since accessing out-of-range elements do not cause errors on compilation, but can cause errors on runtime. The reason for this being allowed will be seen in a later chapter when pointers are introduced*

Multidimensional arrays can be described as "arrays of arrays". For example, a bidimensional array can be imagined as a two-dimensional table made of elements, all of them of a same uniform data type.

int myArr5[3][5] = {{10,2,3,1,9}, {1, 2, 3, 4, 5}, {9, 6, 4, 2, 0}};

Multidimensional arrays are just abstractions for a programmer*.* With the only difference that with multidimensional arrays, the compiler automatically remembers the depth of each imaginary dimension.

|  |  |
| --- | --- |
| **multidimensional array** | **pseudo-multidimensional array** |
| #define WIDTH 5  #define HEIGHT 3  int jimmy [HEIGHT][WIDTH];  int n,m;  int main ()  {  for (n=0; n<HEIGHT; n++)  for (m=0; m<WIDTH; m++)  {  jimmy[n][m]=(n+1)\*(m+1);  }  } | #define WIDTH 5  #define HEIGHT 3  int jimmy [HEIGHT \* WIDTH];  int n,m;  int main ()  {  for (n=0; n<HEIGHT; n++)  for (m=0; m<WIDTH; m++)  {  jimmy[n\*WIDTH+m]=(n+1)\*(m+1);  }  } |

it is not possible to pass the entire block of memory represented by an array to a function directly as an argument. To accept an array as parameter for a function, the parameters can be declared as the array type, but with empty brackets, omitting the actual size of the array.

void procedure (int arg[])

//Only address is passed in an array

void funct1(int arr[], int size){

    for (int i = 0; i < size; i++)

    {

        std::cout << arr[i] << " ";

        arr[i] = i;

    }

}

To accept a multidim array for a function,, the parameter can be declared as

void procedure (base\_type[][depth][depth])

void funct2(int arr[][5], int height){

    for (int i = 0; i < height; i++)

    {

        int j =0;

        while (j<5)

        {

            std::cout << arr[i][j]<< " ";

            j++;

        }

    }

}

***Note****: In a way, passing an array as argument always loses a dimension. The reason behind is that, for historical reasons, arrays cannot be directly copied, and thus what is really passed is a pointer. This is a common source of errors for novice programmers.*

### Library arrays

C++ provides an alternative array type as a standard container. It is a type template (a class template, in fact) defined in header [<array>](https://www.cplusplus.com/%3Carray%3E). They operate in a similar way to built-in arrays, except that they allow being copied (an actually expensive operation that copies the entire block of memory, and thus to use with care) and decay into pointers only when explicitly told to do so.

 std::array<int, 6> myArr6 {1,2,3,4,5,6};

template <std::size\_t SIZE>

void funct3(std::array<int, SIZE> arr){

    for (auto &elem : arr)   //range based loops cannot deal with a pointer.

    {

        std::cout << elem << " ";

    }

}

funct3<myArr6.size()>(myArr6); //Call to funct3 template function

|  |  |
| --- | --- |
| **language built-in array** | **container library array** |
| #include <iostream>  using namespace std;  int main()  {  int myarray[3] = {10,20,30};  for (int i=0; i<3; ++i)  ++myarray[i];  for (int elem : myarray)  cout << elem << '\n';  } | #include <iostream>  #include <array>  using namespace std;  int main()  {  array<int,3> myarray {10,20,30};  for (int i=0; i<myarray.size(); ++i)  ++myarray[i];  for (int elem : myarray)  cout << elem << '\n';  } |

21/03/2021

## Strings

 strings are, in fact, sequences of characters, we can represent them also as plain arrays of elements of a character type.

 The end of strings represented in character sequences is signaled by a special character: the null character, whose literal value can be written as '\0' (backslash, zero).

But arrays of character elements have another way to be initialized: using string literals directly.

Sequences of characters enclosed in double-quotes (") are *literal constants*. And their type is, in fact, a null-terminated array of characters. This means that string literals always have a null character ('\0') automatically appended at the end.

    char myStr[] = {'A', 'n', 'e', 'e', 'k', 'e', 't', '\0'};

    char myStr2[] = "Aneeket";

    std::string myword = "man";

***Note****: once an array of characters is defined, It is not possible to assign values to them later on.*

*Note: Plain arrays with null-terminated sequences of characters are the typical types used in the C language to represent strings (that is why they are also known as C-strings). string literals still always produce null-terminated character sequences, and not string objects.*

*Note: most functions related to strings are overloaded to support both C-string and library strings.*

Arrays have a fixed size that needs to be specified either implicit or explicitly when declared; Hence C-Strings, while strings are simply strings, no size is specified. This is due to the fact that strings have a dynamic size determined during runtime, while the size of arrays is determined on compilation, before the program runs.

***Note****: Null-terminated character sequences can be transformed into strings implicitly, and strings can be transformed into null-terminated character sequences by using either of string's member functions c\_str or data*.

    //Conversion of c-string to string and vice versa:

    std::string myStr3 = myStr; //c-string to string

    const char \*p = myStr3.data(); //string to c-string using member function c\_str() or data()

    // char myStr4[] = myStr3.c\_str();

    //Not allowed as arrays are first need to be initialized and cannot be copied directly.

## Pointers

Variables have been explained as locations in the computer's memory which can be accessed by their identifier (their name). The memory of a computer is like a succession of memory cells, each one byte in size, and each with a unique address. These single-byte memory cells are ordered in a way that allows data representations larger than one byte to occupy memory cells that have consecutive addresses.

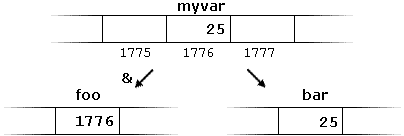
### Address-of operator (&)

When a variable is declared, the memory needed to store its value is assigned a specific location in memory (its memory address).

The address of a variable can be obtained by preceding the name of a variable with an ampersand sign (&), known as *address-of operator*.

    int myInt = 2012;

    auto myIntAd = &myInt



***Note****: The variable that stores the address of another variable (like foo in the previous example) is what in C++ is called a pointer.*

### Dereference operator (\*)

As just seen, a variable which stores the address of another variable is called a *pointer*. Pointers are said to "point to" the variable whose address they store.

An interesting property of pointers is that they can be used to access the variable they point to directly. This is done by preceding the pointer name with the *dereference operator* (\*). The operator itself can be read as "value pointed to by".

    int myInt = 2012;

    auto myIntAd = &myInt;

    int var1 = 10, var2 = 20;

    var1 = var2; //value contained at address of var2 is assigned at var1's address.

    //Addresses remain same for both vars while, value of only var1 changes

    //Ampersand operator outputs memory location, where the variable is stored.

    int var3 = \*myIntAd; //value of var3 equal to value pointed to by myIntAd (Value of myIntAd is treated as address)

*Note: The reference and dereference operators are thus complementary:*

1. *& is the address-of operator, and can be read simply as "address of"*
2. *\* is the dereference operator, and can be read as "value pointed to by"*

### Declaring pointers

The declaration of a pointer needs to include the data type the pointer is going to point to.

    type \* name;

where type is the data type pointed to by the pointer. This type is not the type of the pointer itself, but the type of the data the pointer points to.

    int \* number;

    char \* character;

    double \* decimals;

These are three declarations of pointers. Each one is intended to point to a different data type, but, in fact, all of them are pointers and all of them are likely going to occupy the same amount of space in memory.

***Note:*** *The asterisk (\*) used when declaring a pointer only means that it is a pointer (it is part of its type compound specifier), and should not be confused with the dereference operator seen a bit earlier, but which is also written with an asterisk (\*).*

When this operator precedes the pointer name, the expression refers to the value being pointed, while when a pointer name appears without this operator, it refers to the value of the pointer itself (i.e., the address of what the pointer is pointing to).

***Note****:  remembering to put one asterisk per pointer is enough for most pointer users interested in declaring multiple pointers per statement.*

### Pointers and arrays

arrays work very much like pointers to their first elements, and, actually, an array can always be implicitly converted to the pointer of the proper type.

Array name is pointer to itself

Note: assignment of pointer to array is valid, it assigns address of first element of array to the pointer, but array cannot be assigned to pointer.

 brackets ([]) were explained as specifying the index of an element of the array. Well, in fact these brackets are a dereferencing operator known as *offset operator*. They dereference the variable they follow just as \* does, but they also add the number between brackets to the address being dereferenced.

    int arr[] {1,2,3,4,5};

    int \* myPtr = arr;

    std::cout <<"Pointer: " << myPtr<<std::endl;

    std::cout << "Third value of arr: " << \*(myPtr + 2) <<std::endl;

    myPtr++;

    std::cout <<"Pointer: " << myPtr<<std::endl;

    std::cout << "Second value of arr: " << \*myPtr<<std::endl;

    myPtr = &arr[3];

    std::cout <<"Pointer: " << myPtr<<std::endl;

    std::cout << "Fourth value of arr: " << \*myPtr<<std::endl;

    \*(myPtr+1) = 6;

    std::cout << arr[4] << std::endl;

22/03/2021

### Initializing pointers

When pointers are initialized, what is initialized is the address they point to (i.e., myptr), never the value being pointed.

    int \* myPtr = arr;

Pointers can be initialized either to the address of a variable (such as in the case above), or to the value of another pointer (or array).

    myPtr = &arr[3];

On 32-bit machine sizeof pointer is 32 bits ( 4 bytes), while on 64 bit machine it's 8 byte. Regardless of what data type they are pointing to, they have fixed size.

### Pointer arithmetic

To begin with, only addition and subtraction operations are allowed; the others make no sense in the world of pointers. But both addition and subtraction have a slightly different behavior with pointers, according to the size of the data type to which they point.

when adding one to a pointer, the pointer is made to point to the following element of the same type, and, therefore, the size in bytes of the type it points to is added to the pointer.

    for (auto &elem : arr)

    {

        std::cout << &elem << " : " << elem << std::endl;

    }

Outputs

0x61fdc0 : 1 //int data type which arr’s type is of 4 bytes

0x61fdc4 : 2

0x61fdc8 : 3

0x61fdcc : 4

0x61fdd0 : 6

***Note****: increment and decrement, have higher precedence than prefix operators, such as the dereference operator (\*)*

\*p++   // same as \*(p++): increment pointer, and dereference unincremented address

\*++p   // same as \*(++p): increment pointer, and dereference incremented address

++\*p   // same as ++(\*p): dereference pointer, and increment the value it points to

(\*p)++ // dereference pointer, and post-increment the value it points to

    \*p++ = \*q++;

    //Equivalent to

    \*p = \*q;

    ++p;

    ++q;

### Pointers and consts (How to make variables read only)

Pointers can be used to access a variable by its address, and this access may include modifying the value pointed. But it is also possible to declare pointers that can access the pointed value to read it, but not to modify it. For this, it is enough with qualifying the type pointed to by the pointer as const.

***Note****: a pointer to non-const can be implicitly converted to a pointer to const. But not the other way around! As a safety feature, pointers to const are not implicitly convertible to pointers to non-const.*

// pointers as arguments:

#include <iostream>

using namespace std;

void increment\_all (int\* start, int\* stop)

{

  int \* current = start;

  while (current != stop) {

    ++(\*current);  // increment value pointed

    ++current;     // increment pointer

  }

}

void print\_all (const int\* start, const int\* stop)

{

  const int \* current = start; //value pointed by current are read only

  while (current != stop) {

    cout << \*current << '\n';

    ++current;     // increment pointer

  }

}

int main ()

{

  int numbers[] = {10,20,30};

  increment\_all (numbers,numbers+3);

  print\_all (numbers,numbers+3);

  return 0;

}

*Const and non-const pointers*

    int x;

    int \*p1 = &x;             // non-const pointer to non-const int

    const int \*p2 = &x;       // non-const pointer to const int

    int \*const p3 = &x;       // const pointer to non-const int

    const int \*const p4 = &x; // const pointer to const int

### void pointers

void pointers are pointers that point to a value that has no type (and thus also an undetermined length and undetermined dereferencing properties).

The data pointed to by them cannot be directly dereferenced (which is logical, since we have no type to dereference to), and for that reason, any address in a void pointer needs to be transformed into some other pointer type

### Invalid pointers and null pointers

A *null pointer* is a value that any pointer can take to represent that it is pointing to "nowhere", while a void pointer is a type of pointer that can point to somewhere without a specific type.

int \* p = 0;

int \* q = nullptr;

int \* r = NULL;

a pointer really needs to explicitly point to nowhere, and not just an invalid address. For such cases, there exists a special value that any pointer type can take: the *null pointer value*.

### Pointers to function

The typical use of this is for passing a function as an argument to another function. Pointers to functions are declared with the same syntax as a regular function declaration, except that the name of the function is enclosed between parentheses () and an asterisk (\*) is inserted before the name.

//Why does a pointer to function returns value 1?

1. Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer stores the start of executable code.

2. Unlike normal pointers, we do not allocate de-allocate memory using function pointers.

3. A function’s name can also be used to get functions’ address. For example, in the below program, we have removed address operator ‘&’ in assignment. We have also changed function call by removing \*, the program still works.

4. Like normal pointers, we can have an array of function pointers.

5. Like normal data pointers, a function pointer can be passed as an argument and can also be returned from a function.

#include <iostream>

int divide(int a, int b){

    return a/b;

}

int multiply(int a, int b){

    return a\*b;

}

int operation(int a, int b, int (\*funct) (int , int)){

    int result;

    std::cout << funct <<std::endl;

    result = (\*funct)(a, b);

    return result;

}

int main(int argc, char const \*argv[])

{

    int x = 10;

    int y = 5;

    int (\*division)(int, int) = divide; // pointer with name division points to function divide

    std::cout << division <<std::endl;

    std::cout << divide <<std::endl;

    int res1, res2;

    res1 = operation(x, y, multiply); // A function name is pointer to that function

    res2 = operation(x, y, division); // A pointer to function also points to the function

    std::cout << "Result of x\* y = " << res1 << " and x/y = " << res2;

    return 0;

}

## Dynamic Memory

There is a substantial difference between declaring a normal array and allocating dynamic memory for a block of memory using new. The most important difference is that the size of a regular array needs to be a *constant expression*, and thus its size has to be determined at the moment of designing the program, before it is run, whereas the dynamic memory allocation performed by new allows to assign memory during runtime using any variable value as size.

23/03/2021

### Operators new and new[]

Dynamic memory is allocated using operator new. new is followed by a data type specifier and, if a sequence of more than one element is required, the number of these within brackets []. It returns a pointer to the beginning of the new block of memory allocated.

        int \* myPointer = new int;

        int \* myPointer = new int[size];//allocate multiple elements in sequence

If memory allocation fails then

1. An exception of type bad\_alloc is thrown when the allocation fails.
2. what happens when it is used is that when a memory allocation fails, instead of throwing a bad\_alloc exception or terminating the program, the pointer returned by new is a *null pointer*, and the program continues its execution normally.

        int \* myPointer = new (std::nothrow) int[size];//allocate multiple elements in sequence with nothrow method

#include <iostream>

#include <string>

#include <new>

int main(){

    std::string input;

    while(input != "Exit"){

        int size;

        std::cout << "Enter the number of elements you want to add: ";

        std::cin >> size;

        int \* myPointer = new (std::nothrow) int[size];

        if (myPointer == nullptr) {

            std::cout << "Error: Failed!\n";

            exit;}

        else {

        std::cout << "Success!\n";

        std:: cout << myPointer<<std::endl;

        for (int i=0;i<size ;i++  ) {

            std::cout << "Enter " << i + 1 << "th number: ";

            std::cin >> \*(myPointer + i);

        }

        std::cout << "You entered: ";

        for (int i=0;i<size ;i++  ) {

            std::cout << myPointer[i] << " ";

        }

        delete[] myPointer;}

        std::cout << "\nEnter Exit to exit or anything to continue: ";

        std::cin >> input;

    }

    return 0;

}

24/03/2021

## Data structure

A data structure is group of elements grouped together under one name.

Declaring a structure

    struct type\_name {

    member\_type1 member\_name1;

    member\_type2 member\_name2;

    member\_type3 member\_name3;

    .

    .

    } object\_names;

e.g.

    struct my\_details {

        std::string myName; //members

        int Age; //member

        long long mobile;

    };

its members can be accessed directly. The syntax for that is simply to insert a dot (.) between the object name and the member name.

void printPerson(my\_details person){

    std::cout << "My Age: "<< person.Age

    << "\nMy Name: " << person.myName

    << "\nmobile: "<< person.mobile <<std::endl;

}

### Pointers to structures

struct movies\_t {

  string title;

  int year;

};

movies\_t amovie;

movies\_t \* pmovie; //poimter to object of structure movies\_t type

|  |  |  |
| --- | --- | --- |
| **Expression** | **What is evaluated** | **Equivalent** |
| a.b | Member b of object a |  |
| a->b | Member b of object pointed to by a | (\*a).b |
| \*a.b | Value pointed to by member b of object a | \*(a.b) |

Nesting structures is allowed

struct movies\_t {

  string title;

  int year;

};

struct friends\_t {

  string name;

  string email;

  movies\_t favorite\_movie;

} charlie, maria;

friends\_t \* pfriends = &charlie;

## Typedef and using

Type aliases can be used to reduce the length of long or confusing type names, but they are most useful as tools to abstract programs from the underlying types they use. For example, by using an alias of int to refer to a particular kind of parameter instead of using int directly, it allows for the type to be easily replaced by long (or some other type) in a later version, without having to change every instance where it is used.

typedef char C;

typedef unsigned int WORD;

typedef char \* pChar;

typedef char field [50];

OR

using C = char;

using WORD = unsigned int;

using pChar = char \*;

using field = char [50];

## Union

Unions allow one portion of memory to be accessed as different data types.

It looks like union can be anything as per our choice.

union type\_name {

  member\_type1 member\_name1;

  member\_type2 member\_name2;

  member\_type3 member\_name3;

  .

  .

} object\_names;

Each of these members is of a different data type. But since all of them are referring to the same location in memory, the modification of one of the members will affect the value of all of them. It is not possible to store different values in them in a way that each is independent of the others.

union mix\_t {

  int l;

  struct {

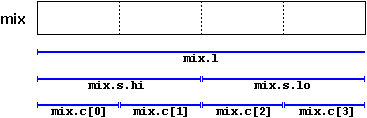
    short hi;

    short lo;

    } s;

  char c[4];

} mix;



***Anonymous unions****: When unions are members of a class (or structure), they can be declared with no name. In this case, they become anonymous unions, and its members are directly accessible from objects by their member names.*

|  |  |
| --- | --- |
| **structure with regular union** | **structure with anonymous union** |
| struct book1\_t {  char title[50];  char author[50];  union {  float dollars;  int yen;  } price;  } book1; | struct book2\_t {  char title[50];  char author[50];  union {  float dollars;  int yen;  };  } book2; |
|  |  |

## Enumerated type (Enum)

Enumerated types are types that are defined with a set of custom identifiers, known as *enumerators*, as possible values. Objects of these *enumerated types* can take any of these enumerators as value.

enum type\_name {

  value1,

  value2,

  value3,

  .

  .

} object\_names;

***Note****: enum creates a whole new data type from scratch without basing it on any other existing type. Values of enumerated types declared with enum are implicitly convertible to an integer type. In fact, the elements of such an enum are always assigned an integer numerical equivalent internally, to which they can be implicitly converted to.*

It is possible to create real enum types that are neither implicitly convertible to int and that neither have enumerator values of type int, but of the enum type itself, thus preserving type safety. They are declared with enum class (or enum struct) instead of just enum:

Each of the enumerator values of an enum class type needs to be scoped into its type.

## Classes

classes are an expanded concept of *data structures*: like data structures, they can contain data members, but they can also contain functions as members.

An object is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.

class class\_name {

  access\_specifier\_1:

    member1;

  access\_specifier\_2:

    member2;

  ...

} object\_names;

An *access specifier* is one of the following three keywords: private, public or protected. These specifiers modify the access rights for the members that follow them:

* private members of a class are accessible only from within other members of the same class (or from their *"friends"*).
* protected members are accessible from other members of the same class (or from their *"friends"*), but also from members of their derived classes.
* Finally, public members are accessible from anywhere where the object is visible.

***Note****: By default, all members of a class declared with the class keyword have private access for all its members.*

class Student{

    std::string firstName;

    std::string lastName;

    int age;

    public:

    // function to initialize object

    void setValues(std::string fName, std::string lName, int Age){

        firstName = fName;

        lastName = lName;

        age = Age;

    }

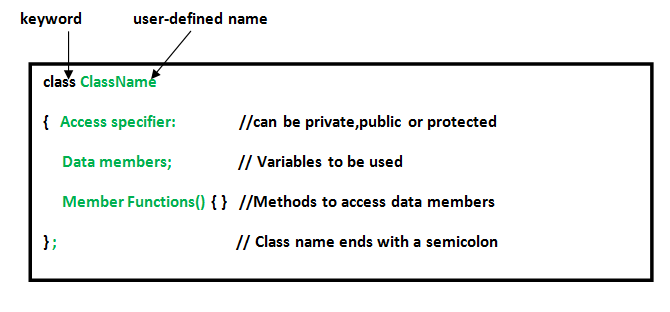
    void printDetails(){

        std::cout << "Name: " << firstName << " " << lastName<<std::endl;

        std::cout << "Age: " << age <<std::endl;

    }

};



### Constructor

A class can include a special function called its *constructor*, which is automatically called whenever a new object of this class is created, allowing the class to initialize member variables or allocate storage.

    std::string firstName;

    std::string lastName;

    int age;

    public:

    //constructor

    Student(std::string fName, std::string lName, int Age){

        firstName = fName;

        lastName = lName;

        age = Age;

    }

This constructor function is declared just like a regular member function, but with a name that matches the class name and without any return type; not even void.

Constructors cannot be called explicitly as if they were regular member functions. They are only executed once, when a new object of that class is created.  
  
***Note****: Neither the constructor prototype declaration (within the class) nor the latter constructor definition, have return values; not even void: Constructors never return values, they simply initialize the object*.

### Constructor overloading

Like any other function, a constructor can also be overloaded with different versions taking different parameters: with a different number of parameters and/or parameters of different types. The compiler will automatically call the one whose parameters match the arguments

    //default constructor

    Student(){

        firstName = "Lorem";

        lastName = "Epsum";

        age = 18;

    }

    //Parameterized constructor

    Student(std::string fName, std::string lName, int Age){

        firstName = fName;

        lastName = lName;

        age = Age;

    }

When an object is initialized without any parameter; default constructor is called.

int main(int argc, char const \*argv[])

{

    //Initialized object

    Student student1("Aneeket", "Lande", 25);

    student1.printDetails();

    //Unintialized object

    Student student2;

    student2.printDetails();

    return 0;

}

Outputs

Name: Aneeket Lande

Age: 25

Name: Lorem Epsum

Age: 18

### Uniform initialization

First, constructors with a single parameter can be called using the variable initialization syntax (an equal sign followed by the argument):  
  
class\_name object\_name = initialization\_value;  
  
More recently, C++ introduced the possibility of constructors to be called using *uniform initialization*, which essentially is the same as the functional form, but using braces ({}) instead of parentheses (()):  
  
class\_name object\_name { value, value, value, ... }

Optionally, this last syntax can include an equal sign before the braces.

### Member initialization in constructors

When a constructor is used to initialize other members, these other members can be initialized directly, without resorting to statements in its body. This is done by inserting, before the constructor's body, a colon (:) and a list of initializations for class members. For example, consider a class with the following declaration:

    class Rectangle {

    int width,height;

  public:

    Rectangle(int,int);

    int area() {return width\*height;}

    };

The constructor for this class could be defined, as usual, as:

Rectangle::Rectangle (int x, int y) { width=x; height=y; }

But it could also be defined using *member initialization* as:

Rectangle::Rectangle (int x, int y) : width(x) { height=y; }

Or even:

Rectangle::Rectangle (int x, int y) : width(x), height(y) { }

27/03/2021

### Overloading Operators

 C++ allows most operators to be overloaded so that their behavior can be defined for just about any type, including classes. Here is a list of all the operators that can be overloaded:

|  |
| --- |
| **Overloadable operators** |
| + - \* / = < > += -= \*= /= << >>  <<= >>= == != <= >= ++ -- % & ^ ! |  ~ &= ^= |= && || %= [] () , ->\* -> new  delete new[] delete[] |

The operator overloads are just regular functions which can have any behavior; there is actually no requirement that the operation performed by that overload bears a relation to the mathematical or usual meaning of the operator, although it is strongly recommended.

***Note****: The parameter expected for a member function overload for operations such as operator+ is naturally the operand to the right hand side of the operator. This is common to all binary operators (those with an operand to its left and one operand to its right).*

|  |  |  |  |
| --- | --- | --- | --- |
| **Expression** | **Operator** | **Member function** | **Non-member function** |
| @a | + - \* & ! ~ ++ -- | A::operator@() | operator@(A) |
| a@ | ++ -- | A::operator@(int) | operator@(A,int) |
| a@b | + - \* / % ^ & | < > == != <= >= << >> && || , | A::operator@(B) | operator@(A,B) |
| a@b | = += -= \*= /= %= ^= &= |= <<= >>= [] | A::operator@(B) | - |
| a(b,c...) | () | A::operator()(B,C...) | - |
| a->b | -> | A::operator->() | - |
| (TYPE) a | TYPE | A::operator TYPE() | - |

Note: Some operators may be overloaded in two forms: either as a member function or as a non-member function.

### The keyword this

The keyword this represents a pointer to the object whose member function is being executed. It is used within a class's member function to refer to the object itself.

It is also frequently used in operator= member functions that return objects by reference.

class CVector

{

private:

    int x, y;

public:

    CVector(int, int);

    ~CVector();

    //returns reference to the object

    CVector operator= (const CVector param){

    x = param.x; y = param.y;

    return \*this;

}

};

### Static members

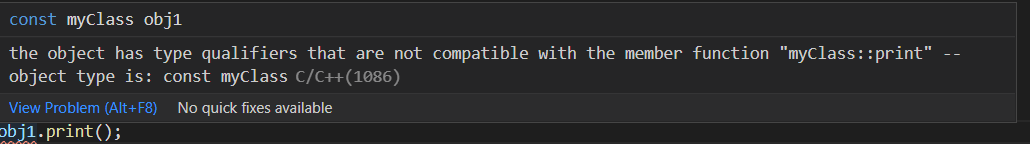
A static data member of a class is also known as a "class variable", because there is only one common variable for all the objects of that same class, sharing the same value: i.e., its value is not different from one object of this class to another.

***Note****: Classes can also have static member functions. These represent the same: members of a class that are common to all object of that class, acting exactly as non-member functions but being accessed like members of the class. Because they are like non-member functions, they cannot access non-static members of the class (neither member variables nor member functions). They neither can use the keyword this.*

***Note****: static members don’t need initialization. They are initialized with the value zero.*

The access to const objects’ data members from outside the class is restricted to read-only, as if all its data members were const for those accessing them from outside the class. Note though, that the constructor is still called and is allowed to initialize and modify these data members

The member functions of a const object can only be called if they are themselves specified as const members

**class myClass

{

private:

    int myInt;

public:

    myClass(int);

    ~myClass();

    void print(); //non const member function

    int square() const {return myInt\* myInt;} //const member function

};

    myClass obj0 (10);

    const myClass obj1 (20);

    obj0.print();

    std::cout << obj0.square();

    // obj1.print(); Not allowed

    std::cout << obj1.square();

Member functions can be overloaded on their constness: i.e., a class may have two member functions with identical signatures except that one is const and the other is not: in this case, the const version is called only when the object is itself const, and the non-const version is called when the object is itself non-const.

    int square() const {

         std::cout << "This is a const function"<< std::endl;

        return myInt\* myInt;} //const member function;

    int square() {

        std::cout << "This is a non-const function"<< std::endl;

        return pow(myInt,2);} //OVERLOADED BASED ON CONST-NESS

### Class templates

#include <iostream>

template <class T>

class MyTriplet{

    T num [3];

    public:

    MyTriplet(T a, T b, T c){

        num[0] = a; num[1] = b; num[2] = c;

    };

    T getMax();

};

/\*In case that a member function is defined outside

the definition of the class template, it shall be

 preceded with the template <...> prefix \*/

template <class T>

T MyTriplet<T>::getMax(){

    T max;

    if (this->num[0] >= this->num[1]) {

        if (this->num[0] >= this->num[2])

            max = this->num[0];

        else

            max = this->num[2];

    } else {

        if (this->num[1] >= this->num[2])

            max = this->num[1];

        else

            max = this->num[2];

    }

    return max;

}

int main(int argc, char const \*argv[])

{

    MyTriplet<int> myInts (100, 200, 300);

    std::cout << myInts.getMax() << std::endl;

    MyTriplet<short> myShorts (1,2, 3);

    std::cout << myShorts.getMax() << std::endl;

    MyTriplet<float> myFloats (10.23, 20.34, 30.45);

    std::cout << myFloats.getMax() << std::endl;

    MyTriplet<double> myDoubles (100.234, 200.345, 300.456);

    std::cout << myDoubles.getMax() << std::endl;

    return 0;

}

### Template specialization

It is possible to define a different implementation for a template when a specific type is passed as template argument. This is called a *template specialization*.

// template specialization

#include <iostream>

using namespace std;

// class template:

template <class T>

class mycontainer {

    T element;

  public:

    mycontainer (T arg) {element=arg;}

    T increase () {return ++element;}

};

// class template specialization:

template <>

class mycontainer <char> {

    char element;

  public:

    mycontainer (char arg) {element=arg;}

    char uppercase ()

    {

      if ((element>='a')&&(element<='z'))

      element+='A'-'a';

      return element;

    }

};

int main () {

  mycontainer<int> myint (7);

  mycontainer<char> mychar ('x');

  cout << myint.increase() << endl;

  cout << mychar.uppercase() << endl;

  return 0;

}

***Note****: When we declare specializations for a template class, we must also define all its members, even those identical to the generic template class, because there is no "inheritance" of members from the generic template to the specialization.*

## Special members

Special member functions are member functions that are implicitly defined as member of classes under certain circumstances.

|  |  |
| --- | --- |
| **Member function** | **typical form for class C:** |
| [Default constructor](http://cplusplus.com/doc/tutorial/classes2/#default_constructor) | C::C(); |
| [Destructor](http://cplusplus.com/doc/tutorial/classes2/#destructor) | C::~C(); |
| [Copy constructor](http://cplusplus.com/doc/tutorial/classes2/#copy_constructor) | C::C (const C&); |
| [Copy assignment](http://cplusplus.com/doc/tutorial/classes2/#copy_assignment) | C& operator= (const C&); |
| [Move constructor](http://cplusplus.com/doc/tutorial/classes2/#move) | C::C (C&&); |
| [Move assignment](http://cplusplus.com/doc/tutorial/classes2/#move) | C& operator= (C&&); |

### Default constructor

The default constructor is the constructor called when objects of a class are declared, but are not initialized with any arguments.

***Note****: But as soon as a class has some constructor taking any number of parameters explicitly declared, the compiler no longer provides an implicit default constructor, and no longer allows the declaration of new objects of that class without arguments. No implicit default constructors is automatically provided.*

### Destructor

Destructors fulfill the opposite functionality of *constructors*: They are responsible for the necessary cleanup needed by a class when its lifetime ends.

 it would be very useful to have a function called automatically at the end of the object's life in charge of releasing this memory. To do this, we use a *destructor*. A destructor is a member function very similar to a *default constructor*: it takes no arguments and returns nothing, not even void.

class Example {

    string\* ptr;

  public:

    // constructors:

    Example() : ptr(new string) {}

    Example (const string& str) : ptr(new string(str)) {}

    // destructor:

    ~Example () {delete ptr;}

    // access content:

    const string& content() const {return \*ptr;}

};

int main () {

  Example foo;

  Example bar ("Example");

  cout << "bar's content: " << bar.content() << '\n';

  return 0;

}

### Copy Constructor

When an object is passed a named object of its own type as argument, its *copy constructor* is invoked in order to construct a copy.

A *copy constructor* is a constructor whose first parameter is of type *reference to the class* itself (possibly const qualified) and which can be invoked with a single argument of this type.

***Internet****:* ***Copy constructors****should not modify the object it is****copying****from which is why the****const****is preferred on the other parameter. Both will work, but the****const****is preferred because it clearly states that the object passed in should not be modified by the function.****const****is for the user only.*

*Ref1:* [*http://cplusplus.com/doc/tutorial/functions/*](http://cplusplus.com/doc/tutorial/functions/)

*Ref2: Efficiency considerations and const references*

If a class has no custom copy nor move constructors (or assignments) defined, an implicit copy constructor is provided. This copy constructor simply performs a copy of its own members. For example, for a class such as:

class MyClass {

  public:

    int a, b; string c;

};

An implicit copy constructor is automatically defined. The definition assumed for this function performs a shallow copy, roughly equivalent to:

MyClass::MyClass(const MyClass& x) : a(x.a), b(x.b), c(x.c) {}

***Note****: shallow copies only copy the members of the class themselves. Performing a shallow copy means that the pointer value is copied, but not the content itself. This is solved by defining a copy constructor which allocates new memory to*

// classes and default constructors

#include <iostream>

#include <string>

using namespace std;

class Example {

    string\* ptr; //pointer defined

    public:

    Example (const string& str) : ptr(new string(str)) {}

    Example() {}

    //destructor

    ~Example(){delete ptr;}

    //shallow copy constructor

    //refers to same data members, also copies pointer values but not the content at that place. Hence pointer points to non-distinct/same content.

    // Example( Example& x): ptr(x.ptr){}

    //Deep copy constructor

//create new copies of data members by dereferencing pointers

    Example( Example& x): ptr(new string(x.content())){}

    //access content returns reference to the string

    const string& content() const {return \*ptr;}

};

int main () {

  Example eg1;

  Example eg2 ("Example");

  eg1 = eg2;  //copy by assignment

  Example eg3 ("Example2");

  Example eg4 (eg2);  // copy object by copy constructor

  cout << "bar's content: " << eg1.content() << '\n';

  cout << "bar's content: " << eg2.content() << '\n';

  cout << "bar's content: " << eg3.content() << '\n';

  cout << "bar's content: " << eg4.content() << '\n';

  return 0;

}

### Copy by Assignment

Objects are not only copied on construction, when they are initialized: They can also be copied on any assignment operation.

The declaration of an object is not an assignment operation, it is just another of the syntaxes to call single-argument constructors.

MyClass foo;

MyClass bar (foo);       // object initialization: copy constructor called

MyClass baz = foo;       // object initialization: copy constructor called

foo = bar;               // object already initialized: copy assignment called

Note that baz is initialized on construction using an *equal sign*, but this is not an assignment operation! (although it may look like one): The declaration of an object is not an assignment operation, it is just another of the syntaxes to call single-argument constructors.  
  
The assignment on foo is an assignment operation. No object is being declared here, but an operation is being performed on an existing object; foo.  
  
The *copy assignment operator* is an overload of operator= which takes a *value* or *reference* of the class itself as parameter. The return value is generally a reference to \*this (although this is not required). For example, for a class MyClass, the *copy assignment* may have the following signature:

MyClass& operator= (const MyClass&);

The *copy assignment operator* is also a *special function* and is also defined implicitly if a class has no custom *copy* nor *move* assignments (nor move constructor) defined.  
  
But again, the *implicit* version performs a *shallow copy* which is suitable for many classes, but not for classes with pointers to objects they handle its storage. In this case, not only the class incurs the risk of deleting the pointed object twice, but the assignment creates memory leaks by not deleting the object pointed by the object before the assignment. These issues could be solved with a *copy assignment* that deletes the previous object and performs a *deep copy*:

Example& operator= (const Example& x) {

  delete ptr;                      // delete currently pointed string

  ptr = new string (x.content());  // allocate space for new string, and copy

  return \*this;

}

Or even better, since its string member is not constant, it could re-utilize the same string object:

Example& operator= (const Example& x) {

  \*ptr = x.content();

  return \*this;

}

### Move constructor and assignment

Similar to copying, moving also uses the value of an object to set the value to another object. But, unlike copying, the content is actually transferred from one object (the source) to the other (the destination): the source loses that content, which is taken over by the destination. This moving only happens when the source of the value is an *unnamed object*.

*Unnamed objects* are objects that are temporary in nature, and thus haven't even been given a name. Typical examples of *unnamed objects* are return values of functions or type-casts.

Using the value of a temporary object such as these to initialize another object or to assign its value, does not really require a copy: the object is never going to be used for anything else, and thus, its value can be *moved into* the destination object.

The move constructor and move assignment are members that take a parameter of type *rvalue reference to the class* itself.

MyClass (MyClass&&);             // move-constructor

MyClass& operator= (MyClass&&);  // move-assignment

An *rvalue reference* is specified by following the type with two ampersands (&&). As a parameter, an *rvalue reference* matches arguments of temporaries of this type.

The concept of moving is most useful for objects that manage the storage they use, such as objects that allocate storage with new and delete. In such objects, copying and moving are really different operations:  
- Copying from A to B means that new memory is allocated to B and then the entire content of A is copied to this new memory allocated for B.  
- Moving from A to B means that the memory already allocated to A is transferred to B without allocating any new storage. It involves simply copying the pointer.

The six *special members functions* described above are members implicitly declared on classes under certain circumstances:

|  |  |  |
| --- | --- | --- |
| **Member function** | **implicitly defined:** | **default definition:** |
| [Default constructor](http://cplusplus.com/doc/tutorial/classes2/#default_constructor) | if no other constructors | does nothing |
| [Destructor](http://cplusplus.com/doc/tutorial/classes2/#destructor) | if no destructor | does nothing |
| [Copy constructor](http://cplusplus.com/doc/tutorial/classes2/#copy_constructor) | if no move constructor and no move assignment | copies all members |
| [Copy assignment](http://cplusplus.com/doc/tutorial/classes2/#copy_assignment) | if no move constructor and no move assignment | copies all members |
| [Move constructor](http://cplusplus.com/doc/tutorial/classes2/#move) | if no destructor, no copy constructor and no copy nor move assignment | moves all members |
| [Move assignment](http://cplusplus.com/doc/tutorial/classes2/#move) | if no destructor, no copy constructor and no copy nor move assignment | moves all members |

Notice how not all *special member functions* are implicitly defined in the same cases. This is mostly due to backwards compatibility with C structures and earlier C++ versions, and in fact some include deprecated cases. Fortunately, each class can select explicitly which of these members exist with their default definition or which are deleted by using the keywords default and delete, respectively. The syntax is either one of:

function\_declaration = default;

function\_declaration = delete;

For example:

// default and delete implicit members

#include <iostream>

using namespace std;

class Rectangle {

    int width, height;

  public:

    Rectangle (int x, int y) : width(x), height(y) {}

    Rectangle() = default;

    Rectangle (const Rectangle& other) = delete;

    int area() {return width\*height;}

};

int main () {

  Rectangle foo;

  Rectangle bar (10,20);

  cout << "bar's area: " << bar.area() << '\n';

  return 0;

}

Here, Rectangle can be constructed either with two int arguments or be *default-constructed* (with no arguments). It cannot however be *copy-constructed* from another Rectangle object, because this function has been deleted. Therefore, assuming the objects of the last example, the following statement would not be valid:

Rectangle baz (foo);

It could, however, be made explicitly valid by defining its copy constructor as:

Rectangle::Rectangle (const Rectangle& other) = default;

Which would be essentially equivalent to:

Rectangle::Rectangle (const Rectangle& other) : width(other.width), height(other.height) {}

Note that, the keyword default does not define a member function equal to the *default constructor* (i.e., where *default constructor* means constructor with no parameters), but equal to the constructor that would be implicitly defined if not deleted.  
  
In general, and for future compatibility, classes that explicitly define one copy/move constructor or one copy/move assignment but not both, are encouraged to specify either delete or default on the other special member functions they don't explicitly define.

30/03/2021

### Friendship and Inheritance

In principle, private and protected members of a class cannot be accessed from outside the same class in which they are declared.

Friends are functions or classes declared with the friend keyword. A non-member function can access the private and protected members of a class if it is declared a *friend* of that class. That is done by including a declaration of this external function within the class, and preceding it with the keyword friend.

05/04/2021

Similar to friend functions, a friend class is a class whose members have access to the private or protected members of another class.

class Square;

class Rectangle {

    int height, width;

    public:

    int area(){

        return height\*width;

    }

    void convert(Square);

};

class Square {

    friend class Rectangle;

    private:

    int side;

    public:

    Square(int a): side(a){};

};

void Rectangle::convert(Square s){

    height = s.side;

    width = s.side;

}

Friendships are never corresponded unless specified: In our example, Rectangle is considered a friend class by Square, but Square is not considered a friend by Rectangle.

Another property of friendships is that they are not transitive: The friend of a friend is not considered a friend unless explicitly specified.

### Inheritance

Classes in C++ can be extended, creating new classes which retain characteristics of the base class. This process, known as inheritance, involves a *base class* and a *derived class*: The *derived class* inherits the members of the *base class*, on top of which it can add its own members.

Classes that are derived from others inherit all the accessible members of the base class. That means that if a base class includes a member A and we derive a class from it with another member called B, the derived class will contain both member A and member B.

class derived\_class\_name: public base\_class\_name

{ /\*...\*/ };

***Note****: This access specifier limits the most accessible level for the members inherited from the base class: The members with a more accessible level are inherited with this level instead, while the members with an equal or more restrictive access level keep their restrictive level in the derived class.*

***Note****: When a class inherits another one, the members of the derived class can access the protected members inherited from the base class, but not its private members.*

***Note****: With protected, all public members of the base class are inherited as protected in the derived class. Conversely, if the most restricting access level is specified (private), all the base class members are inherited as private. (If no access level is specified for the inheritance, the compiler assumes private for classes declared with keyword class and public for those declared with struct.)*

|  |  |  |  |
| --- | --- | --- | --- |
| Access | public | protected | private |
| members of the same class | yes | yes | yes |
| members of derived class | yes | yes | no |
| not members | yes | no | no |

In principle, a publicly derived class inherits access to every member of a base class except:

* its constructors and its destructor
* its assignment operator members (operator=)
* its friends
* its private members

#include <iostream>

using namespace std;

class Polygon{

    protected:

    int height, width;

    public:

    Polygon(){cout << "Polygon no parameters\n";};

    void set\_values(int a, int b){

        height = a; width = b;

    }

};

class Output {

  public:

    static void print (int i);

};

void Output::print (int i) {

  cout << i << '\n';

}

class Rectangle: public Polygon, public Output{

    public:

    Rectangle(){cout << "Rectangle no parameters\n";};//Default constructor with no arguments

    Rectangle(int a, int b){width = a; height = b;cout << "Rectangle int parameters\n";}

    int area(){

        return height\*width;

    }

};

class Triangle: public Polygon, public Output{

    public:

    Triangle(){cout << "Triangle no parameters\n";}

    float area(){

        return height\* (float)width/2;

    }

};

class Square: protected Rectangle{

    public:

    Square(int a){height = a ; width = a;cout << "Square int parameters\n";}

    int area(){ //override area method of rectangle class

        return height\*width;

    }

};

Even though access to the constructors and destructor of the base class is not inherited as such, they are automatically called by the constructors and destructor of the derived class.

int main () {

    Polygon pl1;

    Rectangle rect1;

    Rectangle rect2(10, 20);

    Triangle tri;

    tri.set\_values(5, 13);

    Square sq1(5);

    cout << rect2.area() << endl;

    rect2.print(rect2.area()); //multiple inheritance

    cout << tri.area()<< endl;

    cout <<sq1.area()<< endl;

}

Output:

Polygon no parameters

Polygon no parameters

Rectangle no parameters

Polygon no parameters

Rectangle int parameters

Polygon no parameters

Triangle no parameters

Polygon no parameters

Rectangle no parameters

Sqauare int parameters

200

200

32.5

25

Unless otherwise specified, the constructors of a derived class calls the default constructor of its base classes (i.e., the constructor taking no arguments). Calling a different constructor of a base class is possible, using the same syntax used to initialize member variables in the initialization list:  
  
derived\_constructor\_name (parameters) : base\_constructor\_name (parameters) {...}

class Mother {

  public:

    Mother ()

      { cout << "Mother: no parameters\n"; }

    Mother (int a)

      { cout << "Mother: int parameter\n"; }

};

class Daughter : public Mother {

  public:

    Daughter (int a)

      { cout << "Daughter: int parameter\n\n"; }

};

class Son : public Mother {

  public:

    Son (int a) : Mother (a)

      { cout << "Son: int parameter\n\n"; }

};

int main () {

    Daughter Ananya(0);

    Son Rusell(0);

  return 0;

}

Mother: no parameters

Daughter: int parameter

Mother: int parameter

Son: int parameter

## Polymorphism

One of the key features of class inheritance is that a pointer to a derived class is type-compatible with a pointer to its base class.

### Virtual members

A virtual member is a member function that can be redefined in a derived class, while preserving its calling properties through references.

// virtual members

#include <iostream>

using namespace std;

class Polygon {

  protected:

    int width, height;

  public:

    void set\_values (int a, int b)

      { width=a; height=b; }

    virtual int area ()

      { return 0; }

};

class Rectangle: public Polygon {

  public:

    int area ()

      { return width \* height; }

};

class Triangle: public Polygon {

  public:

    int area ()

      { return (width \* height / 2); }

};

int main () {

  Rectangle rect;

  Triangle trgl;

  Polygon poly;

  Polygon \* ppoly1 = &rect;

  Polygon \* ppoly2 = &trgl;

  Polygon \* ppoly3 = &poly;

  ppoly1->set\_values (4,5);

  ppoly2->set\_values (4,5);

  ppoly3->set\_values (4,5);

  cout << ppoly1->area() << '\n';

  cout << ppoly2->area() << '\n';

  cout << ppoly3->area() << '\n';

  return 0;

}

The member function area has been declared as virtual in the base class because it is later redefined in each of the derived classes. Non-virtual members can also be redefined in derived classes, but non-virtual members of derived classes cannot be accessed through a reference of the base class: i.e., if virtual is removed from the declaration of area in the example above, all three calls to area would return zero, because in all cases, the version of the base class would have been called instead.

 What the virtual keyword does is to allow a member of a derived class with the same name as one in the base class to be appropriately called from a pointer, and more precisely when the type of the pointer is a pointer to the base class that is pointing to an object of the derived class, as in the above example.

A class that declares or inherits a virtual function is called a *polymorphic class*.

## Type Conversions

Standard conversions affect fundamental data types, and allow the conversions between numerical types (short to int, int to float, double to int...), to or from bool, and some pointer conversions.

***Note****: Converting to int from some smaller integer type, or to double from float is known as promotion, and is guaranteed to produce the exact same value in the destination type.*

Other conversions between arithmetic types may not always be able to represent the same value exactly:

* If a negative integer value is converted to an unsigned type, the resulting value corresponds to its 2's complement bitwise representation (i.e., -1 becomes the largest value representable by the type, -2 the second largest, ...).
* The conversions from/to bool consider false equivalent to *zero* (for numeric types) and to *null pointer* (for pointer types); true is equivalent to all other values and is converted to the equivalent of 1.
* If the conversion is from a floating-point type to an integer type, the value is truncated (the decimal part is removed). If the result lies outside the range of representable values by the type, the conversion causes *undefined behavior*.
* Otherwise, if the conversion is between numeric types of the same kind (integer-to-integer or floating-to-floating), the conversion is valid, but the value is *implementation-specific* (and may not be portable).

Some of these conversions may imply a loss of precision, which the compiler can signal with a warning. This warning can be avoided with an explicit conversion.

For non-fundamental types, arrays and functions implicitly convert to pointers, and pointers in general allow the following conversions:

* *Null pointers* can be converted to pointers of any type
* Pointers to any type can be converted to void pointers.
* Pointer *upcast*: pointers to a derived class can be converted to a pointer of an *accessible* and *unambiguous* base class, without modifying its const or volatile qualification

06/04/2021

### Implicit conversions with classes

* **Single-argument constructors:** allow implicit conversion from a particular type to initialize an object.
* **Assignment operator:** allow implicit conversion from a particular type on assignments.
* **Type-cast operator:** allow implicit conversion to a particular type.

// implicit conversion of classes:

#include <iostream>

using namespace std;

class A {};

class B {

public:

  // conversion from A (constructor):

  B (const A& x) {}

  // conversion from A (assignment):

  B& operator= (const A& x) {return \*this;}

  // conversion to A (type-cast operator)

  operator A() {return A();}

};

int main ()

{

  A foo;

  B bar = foo;    // calls constructor

  bar = foo;      // calls assignment

  foo = bar;      // calls type-cast operator

  return 0;

}

### Type casting

 Many conversions, specially those that imply a different interpretation of the value, require an explicit conversion, known in C++ as *type-casting*.

    double x = 10.3;

    int y;

    y = int (x);    // functional notation

    y = (int) x;    // c-like cast notation

The functionality of these generic forms of type-casting is enough for most needs with fundamental data types. However, these operators can be applied indiscriminately on classes and pointers to classes, which can lead to code that -while being syntactically correct- can cause runtime errors.

#include <iostream>

class MyClass1{

    double a,b;

    public:

    MyClass1(double i, double j): a(i), b(j){}

    MyClass1(){}

};

class MyClass2{

    int x,y;

    public:

    MyClass2 (int var1, int var2) : x(var1), y(var2){ }

    int result(){return x\*y;}

};

int main(int argc, char const \*argv[])

{

    //explicit type casting

    MyClass1 obj1(3,5);

    MyClass1 obj2;

    MyClass2 \* pmc2;

    //Go to address of obj1 and explicitly make pmc2 to point towards obj1 contents

    pmc2 = (MyClass2\*) &obj1;

    std::cout << pmc2->result()<<std::endl;

    pmc2 = (MyClass2\*) &obj2;

    std::cout << pmc2->result();

    return 0;

}

Unrestricted explicit type-casting allows to convert any pointer into any other pointer type, independently of the types they point to. The subsequent call to member result will produce either a run-time error or some other unexpected results.

In order to control these types of conversions between classes, we have four specific casting operators: dynamic\_cast, reinterpret\_cast, static\_cast and const\_cast. Their format is to follow the new type enclosed between angle-brackets (<>) and immediately after, the expression to be converted between parentheses.

    dynamic\_cast <new\_type> (expression)

    reinterpret\_cast <new\_type> (expression)

    static\_cast <new\_type> (expression)

    const\_cast <new\_type> (expression)

The traditional type-casting equivalents to these expressions would be:

(new\_type) expression

new\_type (expression)

07/04/2021

### dynamic\_cast

dynamic\_cast can only be used with pointers and references to classes (or with void\*). Its purpose is to ensure that the result of the type conversion points to a valid complete object of the destination pointer type.  
  
This naturally includes *pointer upcast* (converting from pointer-to-derived to pointer-to-base), in the same way as allowed as an *implicit conversion*.  
  
But dynamic\_cast can also *downcast* (convert from pointer-to-base to pointer-to-derived) polymorphic classes (those with virtual members) if -and only if- the pointed object is a valid complete object of the target type.

#include <iostream>

#include <exception>

class Base { virtual void Dummmy(){}};

class Derived: public Base { int a;};

main(){

    try {

        Base \* pba = new Derived;

        Base \* pbb = new Base;

        Derived \* pda = new Derived;

        Base \* pb;

        Derived \* pd;

        //downcast: from pointer to base class pointer to derived class

        pd = dynamic\_cast <Derived\*> (pba);

        if (pd==0) std::cout << "Null pointer on first type-cast.\n";

        pd = dynamic\_cast <Derived\*> (pbb);

        if (pd==0) std::cout << "Null pointer on Second type-cast.\n";

        //upcast: from pointer to derived class to pointer to base class

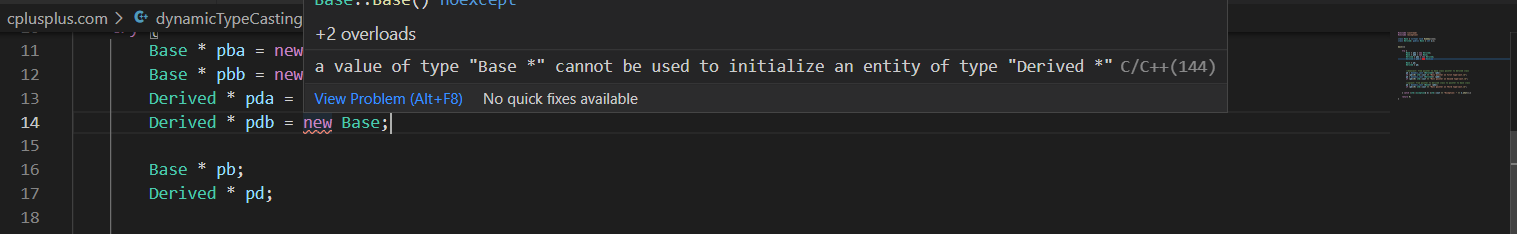
        pb = dynamic\_cast <Base\*> (pda);

        if (pb==0) std::cout << "Null pointer on Third type-cast.\n";

    } catch (std::exception& e) {std::cout << "Exception: " << e.what();}

    return 0;

}



he code above tries to perform two dynamic casts from pointer objects of type Base\* (pba and pbb) to a pointer object of type Derived\*, but only the first one is successful. Notice their respective initializations:

        Base \* pba = new Derived;

        Base \* pbb = new Base;

Even though both are pointers of type Base\*, pba actually points to an object of type Derived, while pbb points to an object of type Base. Therefore, when their respective type-casts are performed using dynamic\_cast, pba is pointing to a full object of class Derived, whereas pbb is pointing to an object of class Base, which is an incomplete object of class Derived.  
  
When dynamic\_cast cannot cast a pointer because it is not a complete object of the required class -as in the second conversion in the previous example- it returns a *null pointer* to indicate the failure. If dynamic\_cast is used to convert to a reference type and the conversion is not possible, an exception of type bad\_cast is thrown instead.  
  
dynamic\_cast can also perform the other implicit casts allowed on pointers: casting null pointers between pointers types (even between unrelated classes), and casting any pointer of any type to a void\* pointer.

## Exceptions

Exceptions provide a way to react to exceptional circumstances (like runtime errors) in programs by transferring control to special functions called *handlers*.

To catch exceptions, a portion of code is placed under exception inspection. This is done by enclosing that portion of code in a *try-block*. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored.

An exception is thrown by using the throw keyword from inside the try block. Exception handlers are declared with the keyword catch, which must be placed immediately after the try block.

// exceptions

#include <iostream>

using namespace std;

int main () {

  try

  {

    throw 125;

  }

  catch (int e)

  {

    cout << "An exception occurred. Exception Nr. " << e << '\n';

  }

  return 0;

}

A throw expression accepts one parameter, which is passed as an argument to the exception handler.  
  
The exception handler is declared with the catch keyword immediately after the closing brace of the try block. The syntax for catch is similar to a regular function with one parameter. The type of this parameter is very important, since the type of the argument passed by the throw expression is checked against it, and only in the case they match, the exception is caught by that handler.  
  
Multiple handlers (i.e., catch expressions) can be chained; each one with a different parameter type. Only the handler whose argument type matches the type of the exception specified in the throw statement is executed.

try {

  // code here

}

catch (int param) { cout << "int exception"; }

catch (char param) { cout << "char exception"; }

catch (...) { cout << "default exception"; }

If an ellipsis (...) is used as the parameter of catch, that handler will catch any exception no matter what the type of the exception thrown. This can be used as a default handler that catches all exceptions not caught by other handlers.

***Note****: After an exception has been handled the program, execution resumes after the try-catch block, not after the throw statement!.*

It is also possible to nest try-catch blocks within more external try blocks. In these cases, we have the possibility that an internal catch block forwards the exception to its external level.

try {

  try {

      // code here

  }

  catch (int n) {

      throw;

  }

}

catch (...) {

  cout << "Exception occurred";

}

### Exception specification

Older code may contain *dynamic exception specifications*. They are now deprecated in C++, but still supported. A *dynamic exception specification* follows the declaration of a function, appending a throw specifier to it. For example:

double myfunction (char param) throw (int);

This declares a function called myfunction, which takes one argument of type char and returns a value of type double. If this function throws an exception of some type other than int, the function calls [std::unexpected](http://cplusplus.com/unexpected) instead of looking for a handler or calling [std::terminate](http://cplusplus.com/terminate).  
  
If this throw specifier is left empty with no type, this means that [std::unexpected](http://cplusplus.com/unexpected) is called for any exception. Functions with no throw specifier (regular functions) never call [std::unexpected](http://cplusplus.com/unexpected), but follow the normal path of looking for their exception handler.

int myfunction (int param) throw(); // all exceptions call unexpected

int myfunction (int param);         // normal exception handling

### Standard exceptions

The C++ Standard library provides a base class specifically designed to declare objects to be thrown as exceptions. It is called std::exception and is defined in the <exception> header. This class has a virtual member function called what that returns a null-terminated character sequence (of type char \*) and that can be overwritten in derived classes to contain some sort of description of the exception.

// using standard exceptions

#include <iostream>

#include <exception>

using namespace std;

class myexception: public exception

{

  virtual const char\* what() const throw()

  {

    return "My exception happened";

  }

} myex;

int main () {

  try

  {

    throw myex;

  }

  catch (exception& e)

  {

    cout << e.what() << '\n';

  }

  return 0;

}

We have placed a handler that catches exception objects by reference (notice the ampersand & after the type), therefore this catches also classes derived from exception, like our myex object of type myexception.

ll exceptions thrown by components of the C++ Standard library throw exceptions derived from this exception class. These are:

|  |  |
| --- | --- |
| **exception** | **description** |
| [bad\_alloc](http://cplusplus.com/bad_alloc) | thrown by new on allocation failure |
| [bad\_cast](http://cplusplus.com/bad_cast) | thrown by dynamic\_cast when it fails in a dynamic cast |
| [bad\_exception](http://cplusplus.com/bad_exception) | thrown by certain dynamic exception specifiers |
| [bad\_typeid](http://cplusplus.com/bad_typeid) | thrown by typeid |
| [bad\_function\_call](http://cplusplus.com/bad_function_call) | thrown by empty [function](http://cplusplus.com/function) objects |
| [bad\_weak\_ptr](http://cplusplus.com/bad_weak_ptr) | thrown by [shared\_ptr](http://cplusplus.com/shared_ptr) when passed a bad [weak\_ptr](http://cplusplus.com/weak_ptr) |

lso deriving from exception, header [<exception>](http://cplusplus.com/%3Cexception%3E) defines two generic exception types that can be inherited by custom exceptions to report errors:

|  |  |
| --- | --- |
| **exception** | **description** |
| [logic\_error](http://cplusplus.com/logic_error) | error related to the internal logic of the program |
| [runtime\_error](http://cplusplus.com/runtime_error) | error detected during runtime |

### Preprocessor directive

Preprocessor directives are lines included in the code of programs preceded by a hash sign (#). These lines are not program statements but directives for the *preprocessor*. The preprocessor examines the code before actual compilation of code begins and resolves all these directives before any code is actually generated by regular statements.  
  
These *preprocessor directives* extend only across a single line of code. As soon as a newline character is found, the preprocessor directive is ends. No semicolon (;) is expected at the end of a preprocessor directive. The only way a preprocessor directive can extend through more than one line is by preceding the newline character at the end of the line by a backslash (\).

### macro definitions (#define, #undef)

To define preprocessor macros we can use #define. Its syntax is:  
  
#define identifier replacement  
  
When the preprocessor encounters this directive, it replaces any occurrence of identifier in the rest of the code by replacement. This replacement can be an expression, a statement, a block or simply anything. The preprocessor does not understand C++ proper, it simply replaces any occurrence of identifier by replacement.

#define TABLE\_SIZE 100

int table1[TABLE\_SIZE];

int table2[TABLE\_SIZE];

#define can work also with parameters to define function macros:

#define getmax(a,b) a>b?a:b

This would replace any occurrence of getmax followed by two arguments by the replacement expression, but also replacing each argument by its identifier, exactly as you would expect if it was a function:

// function macro

#include <iostream>

using namespace std;

#define getmax(a,b) ((a)>(b)?(a):(b))

int main()

{

  int x=5, y;

  y= getmax(x,2);

  cout << y << endl;

  cout << getmax(7,x) << endl;

  return 0;

}

Defined macros are not affected by block structure. A macro lasts until it is undefined with the #undef preprocessor directive:

#define TABLE\_SIZE 100

int table1[TABLE\_SIZE];

#undef TABLE\_SIZE

#define TABLE\_SIZE 200

int table2[TABLE\_SIZE];

Function macro definitions accept two special operators (# and ##) in the replacement sequence:  
The operator #, followed by a parameter name, is replaced by a string literal that contains the argument passed (as if enclosed between double quotes):

#define str(x) #x

cout << str(test);

The operator ## concatenates two arguments leaving no blank spaces between them:

#define glue(a,b) a ## b

glue(c,out) << "test";

This would also be translated into:

cout << "test";

Because preprocessor replacements happen before any C++ syntax check, macro definitions can be a tricky feature. But, be careful: code that relies heavily on complicated macros become less readable, since the syntax expected is on many occasions different from the normal expressions programmers expect in C++.

### Conditional inclusions (#ifdef, #ifndef, #if, #endif, #else and #elif)

These directives allow to include or discard part of the code of a program if a certain condition is met.  
  
#ifdef allows a section of a program to be compiled only if the macro that is specified as the parameter has been defined, no matter which its value is. For example:

#ifdef TABLE\_SIZE

int table[TABLE\_SIZE];

#endif

#ifndef serves for the exact opposite: the code between #ifndef and #endif directives is only compiled if the specified identifier has not been previously defined. For example:

#ifndef TABLE\_SIZE

#define TABLE\_SIZE 100

#endif

int table[TABLE\_SIZE];

In this case, if when arriving at this piece of code, the TABLE\_SIZE macro has not been defined yet, it would be defined to a value of 100. If it already existed it would keep its previous value since the #define directive would not be executed.

#if TABLE\_SIZE>200

#undef TABLE\_SIZE

#define TABLE\_SIZE 200

#elif TABLE\_SIZE<50

#undef TABLE\_SIZE

#define TABLE\_SIZE 50

#else

#undef TABLE\_SIZE

#define TABLE\_SIZE 100

#endif

int table[TABLE\_SIZE];

## Input/ Output operations on files

### Open a file

The first operation generally performed on an object of one of these classes is to associate it to a real file. This procedure is known as to *open a file*. An open file is represented within a program by a *stream* (i.e., an object of one of these classes; in the previous example, this was myfile) and any input or output operation performed on this stream object will be applied to the physical file associated to it.  
  
In order to open a file with a stream object we use its member function open:  
  
open (filename, mode);  
  
Where filename is a string representing the name of the file to be opened, and mode is an optional parameter with a combination of the following flags:

|  |  |
| --- | --- |
| ios::in | Open for input operations. |
| ios::out | Open for output operations. |
| ios::binary | Open in binary mode. |
| ios::ate | Set the initial position at the end of the file. If this flag is not set, the initial position is the beginning of the file. |
| ios::app | All output operations are performed at the end of the file, appending the content to the current content of the file. |
| ios::trunc | If the file is opened for output operations and it already existed, its previous content is deleted and replaced by the new one. |

All these flags can be combined using the bitwise operator OR (|). For example, if we want to open the file example.bin in binary mode to add data we could do it by the following call to member function open:

ofstream myfile;

myfile.open ("example.bin", ios::out | ios::app | ios::binary);

Each of the open member functions of classes ofstream, ifstream and fstream has a default mode that is used if the file is opened without a second argument:

|  |  |
| --- | --- |
| **class** | **default mode parameter** |
| ofstream | ios::out |
| ifstream | ios::in |
| fstream | ios::in | ios::out |

For ifstream and ofstream classes, ios::in and ios::out are automatically and respectively assumed, even if a mode that does not include them is passed as second argument to the open member function (the flags are combined).  
  
For fstream, the default value is only applied if the function is called without specifying any value for the mode parameter. If the function is called with any value in that parameter the default mode is overridden, not combined.  
  
File streams opened in *binary mode* perform input and output operations independently of any format considerations. Non-binary files are known as *text files*, and some translations may occur due to formatting of some special characters (like newline and carriage return characters).  
  
Since the first task that is performed on a file stream is generally to open a file, these three classes include a constructor that automatically calls the open member function and has the exact same parameters as this member. Therefore, we could also have declared the previous myfile object and conduct the same opening operation in our previous example by writing:

ofstream myfile ("example.bin", ios::out | ios::app | ios::binary);

Combining object construction and stream opening in a single statement. Both forms to open a file are valid and equivalent.  
  
To check if a file stream was successful opening a file, you can do it by calling to member is\_open. This member function returns a bool value of true in the case that indeed the stream object is associated with an open file, or false otherwise:

if (myfile.is\_open()) { /\* ok, proceed with output \*/ }

### Closing a file

When we are finished with our input and output operations on a file we shall close it so that the operating system is notified and its resources become available again. For that, we call the stream's member function close. This member function takes flushes the associated buffers and closes the file:

    myfile.close();

Once this member function is called, the stream object can be re-used to open another file, and the file is available again to be opened by other processes.  
  
In case that an object is destroyed while still associated with an open file, the destructor automatically calls the member function close.

### Text files

Text file streams are those where the ios::binary flag is not included in their opening mode. These files are designed to store text and thus all values that are input or output from/to them can suffer some formatting transformations, which do not necessarily correspond to their literal binary value.  
  
Writing operations on text files are performed in the same way we operated with cout:

// writing on a text file

#include <iostream>

#include <fstream>

using namespace std;

int main () {

  ofstream myfile ("example.txt");

  if (myfile.is\_open())

  {

    myfile << "This is a line.\n";

    myfile << "This is another line.\n";

    myfile.close();

  }

  else cout << "Unable to open file";

  return 0;

}

Reading from a file can also be performed in the same way that we did with cin:

// reading a text file

#include <iostream>

#include <fstream>

#include <string>

using namespace std;

int main () {

  string line;

  ifstream myfile ("example.txt");

  if (myfile.is\_open())

  {

    while ( getline (myfile,line) )

    {

      cout << line << '\n';

    }

    myfile.close();

  }

  else cout << "Unable to open file";

  return 0;

}

This last example reads a text file and prints out its content on the screen. We have created a while loop that reads the file line by line, using [getline](http://cplusplus.com/getline). The value returned by [getline](http://cplusplus.com/getline) is a reference to the stream object itself, which when evaluated as a boolean expression (as in this while-loop) is true if the stream is ready for more operations, and false if either the end of the file has been reached or if some other error occurred.

### Checking state flags

The following member functions exist to check for specific states of a stream (all of them return a bool value):

bad()

Returns true if a reading or writing operation fails. For example, in the case that we try to write to a file that is not open for writing or if the device where we try to write has no space left.

fail()

Returns true in the same cases as bad(), but also in the case that a format error happens, like when an alphabetical character is extracted when we are trying to read an integer number.

eof()

Returns true if a file open for reading has reached the end.

good()

It is the most generic state flag: it returns false in the same cases in which calling any of the previous functions would return true. Note that good and bad are not exact opposites (good checks more state flags at once).

The member function clear() can be used to reset the state flags.

### get and put stream positioning

All i/o streams objects keep internally -at least- one internal position:  
  
ifstream, like istream, keeps an internal *get position* with the location of the element to be read in the next input operation.  
  
ofstream, like ostream, keeps an internal *put position* with the location where the next element has to be written.  
  
Finally, fstream, keeps both, the *get* and the *put position*, like iostream.  
  
These internal stream positions point to the locations within the stream where the next reading or writing operation is performed. These positions can be observed and modified using the following member functions:

**tellg() and tellp()**

These two member functions with no parameters return a value of the member type streampos, which is a type representing the current *get position* (in the case of tellg) or the *put position* (in the case of tellp).

**seekg() and seekp()**

These functions allow to change the location of the *get* and *put positions*. Both functions are overloaded with two different prototypes. The first form is:

seekg ( position );

seekp ( position );

Using this prototype, the stream pointer is changed to the absolute position position (counting from the beginning of the file). The type for this parameter is streampos, which is the same type as returned by functions tellg and tellp.  
  
The other form for these functions is:

seekg ( offset, direction );

seekp ( offset, direction );

Using this prototype, the *get* or *put position* is set to an offset value relative to some specific point determined by the parameter direction. offset is of type streamoff. And direction is of type seekdir, which is an *enumerated type* that determines the point from where offset is counted from, and that can take any of the following values:

|  |  |
| --- | --- |
| ios::beg | offset counted from the beginning of the stream |
| ios::cur | offset counted from the current position |
| ios::end | offset counted from the end of the stream |

The following example uses the member functions we have just seen to obtain the size of a file:

// obtaining file size

#include <iostream>

#include <fstream>

using namespace std;

int main () {

  streampos begin,end;

  ifstream myfile ("example.bin", ios::binary);

  begin = myfile.tellg();

  myfile.seekg (0, ios::end);

  end = myfile.tellg();

  myfile.close();

  cout << "size is: " << (end-begin) << " bytes.\n";

  return 0;

}

Notice the type we have used for variables begin and end:

    streampos size;

streampos is a specific type used for buffer and file positioning and is the type returned by file.tellg(). Values of this type can safely be subtracted from other values of the same type, and can also be converted to an integer type large enough to contain the size of the file.  
  
These stream positioning functions use two particular types: streampos and streamoff. These types are also defined as member types of the stream class:

|  |  |  |
| --- | --- | --- |
| **Type** | **Member type** | **Description** |
| [streampos](http://cplusplus.com/streampos) | [ios::pos\_type](http://cplusplus.com/ios#types) | Defined as [fpos<mbstate\_t>](http://cplusplus.com/fpos). It can be converted to/from [streamoff](http://cplusplus.com/streamoff) and can be added or subtracted values of these types. |
| [streamoff](http://cplusplus.com/streamoff) | [ios::off\_type](http://cplusplus.com/ios#types) | It is an alias of one of the fundamental integral types (such as int or long long). |

Each of the member types above is an alias of its non-member equivalent (they are the exact same type). It does not matter which one is used. The member types are more generic, because they are the same on all stream objects (even on streams using exotic types of characters), but the non-member types are widely used in existing code for historical reasons.

### Binary files

For binary files, reading and writing data with the extraction and insertion operators (<< and >>) and functions like getline is not efficient, since we do not need to format any data and data is likely not formatted in lines.  
  
File streams include two member functions specifically designed to read and write binary data sequentially: write and read. The first one (write) is a member function of ostream (inherited by ofstream). And read is a member function of istream (inherited by ifstream). Objects of class fstream have both. Their prototypes are:  
  
write ( memory\_block, size );  
read ( memory\_block, size );  
  
Where memory\_block is of type char\* (pointer to char), and represents the address of an array of bytes where the read data elements are stored or from where the data elements to be written are taken. The size parameter is an integer value that specifies the number of characters to be read or written from/to the memory block.

// reading an entire binary file

#include <iostream>

#include <fstream>

using namespace std;

int main () {

  streampos size;

  char \* memblock;

  ifstream file ("example.bin", ios::in|ios::binary|ios::ate);

  if (file.is\_open())

  {

    size = file.tellg();

    memblock = new char [size];

    file.seekg (0, ios::beg);

    file.read (memblock, size);

    file.close();

    cout << "the entire file content is in memory";

    delete[] memblock;

  }

  else cout << "Unable to open file";

  return 0;

}

In this example, the entire file is read and stored in a memory block. Let's examine how this is done:  
  
First, the file is open with the ios::ate flag, which means that the get pointer will be positioned at the end of the file. This way, when we call to member tellg(), we will directly obtain the size of the file.  
  
Once we have obtained the size of the file, we request the allocation of a memory block large enough to hold the entire file:

    memblock = new char [size];

Right after that, we proceed to set the *get position* at the beginning of the file (remember that we opened the file with this pointer at the end), then we read the entire file, and finally close it:

    file.seekg (0, ios::beg);

    file.read (memblock, size);

    file.close();

At this point we could operate with the data obtained from the file. But our program simply announces that the content of the file is in memory and then finishes

### Buffers and Synchronization

When we operate with file streams, these are associated to an internal buffer object of type streambuf. This buffer object may represent a memory block that acts as an intermediary between the stream and the physical file. For example, with an ofstream, each time the member function put (which writes a single character) is called, the character may be inserted in this intermediate buffer instead of being written directly to the physical file with which the stream is associated.  
  
The operating system may also define other layers of buffering for reading and writing to files.  
  
When the buffer is flushed, all the data contained in it is written to the physical medium (if it is an output stream). This process is called *synchronization* and takes place under any of the following circumstances:

* **When the file is closed:** before closing a file, all buffers that have not yet been flushed are synchronized and all pending data is written or read to the physical medium.
* **When the buffer is full:** Buffers have a certain size. When the buffer is full it is automatically synchronized.
* **Explicitly, with manipulators:** When certain manipulators are used on streams, an explicit synchronization takes place. These manipulators are: [flush](http://cplusplus.com/flush) and [endl](http://cplusplus.com/endl).
* **Explicitly, with member function sync():** Calling the stream's member function sync() causes an immediate synchronization. This function returns an int value equal to -1 if the stream has no associated buffer or in case of failure. Otherwise (if the stream buffer was successfully synchronized) it returns 0.

# Notes from internet

## Format specifiers for printf

|  |
| --- |
| <https://codeforwin.org/2015/05/list-of-all-format-specifiers-in-c-programming.html> |
| **Format specifier** | **Description** | **Supported data types** |
| %c | Character | char unsigned char |
| %d | Signed Integer | short unsigned short int long |
| %e or %E | Scientific notation of float values | float double |
| %f | Floating point | float |
| %g or %G | Similar as %e or %E | float double |
| %hi | Signed Integer(Short) | short |
| %hu | Unsigned Integer(Short) | unsigned short |
| %i | Signed Integer | short unsigned short int long |
| %l or %ld or %li | Signed Integer | long |
| %lf | Floating point | double |
| %Lf | Floating point | long double |
| %lu | Unsigned integer | unsigned int unsigned long |
| %lli, %lld | Signed Integer | long long |
| %llu | Unsigned Integer | unsigned long long |
| %o | Octal representation of Integer. | short unsigned short int unsigned int long |
| %p | Address of pointer to void void \* | void \* |
| %s | String | char \* |
| %u | Unsigned Integer | unsigned int unsigned long |
| %x or %X | Hexadecimal representation of Unsigned Integer | short unsigned short int unsigned int long |
| %n | Prints nothing |  |
| %% | Prints % character |  |

Stack vs Heap memory

## Stack vs Heap Memory Allocation

### ****Stack Allocation****

 The allocation happens on contiguous blocks of memory. We call it a stack memory allocation because the allocation happens in the function call stack. The size of memory to be allocated is known to the compiler and whenever a function is called, its variables get memory allocated on the stack. And whenever the function call is over, the memory for the variables is deallocated. This all happens using some predefined routines in the compiler. A programmer does not have to worry about memory allocation and deallocation of stack variables. This kind of memory allocation also known as Temporary memory allocation because as soon as the method finishes its execution all the data belongs to that method flushes out from the stack automatically. Means, any value stored in the stack memory scheme is accessible as long as the method hasn’t completed its execution and currently in running state.

**Key Points:**

* It’s a temporary memory allocation scheme where the data members are accessible only if the method( ) that contained them is currently is running.
* It allocates or deallocates the memory automatically as soon as the corresponding method completes its execution.
* We receive the corresponding error Java. lang. StackOverFlowError by JVM, If the stack memory is filled completely.
* Stack memory allocation is considered safer as compared to heap memory allocation because the data stored can only be access by owner thread.
* Memory allocation and deallocation is faster as compared to Heap-memory allocation.
* Stack-memory has less storage space as compared to Heap-memory.

int main()

{

   // All these variables get memory

   // allocated on stack

   int a;

   int b[10];

   int n = 20;

   int c[n];

}

### ****Heap Allocation****

The memory is allocated during the execution of instructions written by programmers. Note that the name heap has nothing to do with the heap data structure. It is called heap because it is a pile of memory space available to programmers to allocated and de-allocate. Every time when we made an object it always creates in Heap-space and the referencing information to these objects are always stored in Stack-memory. Heap memory allocation isn’t as safe as Stack memory allocation was because the data stored in this space is accessible or visible to all threads. If a programmer does not handle this memory well, a [memory leak](https://www.geeksforgeeks.org/what-is-memory-leak-how-can-we-avoid/) can happen in the program.

**The Heap-memory allocation is further divided into three categories:-** These three categories help us to prioritize the data(Objects) to be stored in the Heap-memory or in the Garbage collection.

* **Young Generation –** It’s the portion of the memory where all the new data(objects) are made to allocate the space and whenever this memory is completely filled then the rest of the data is stored in Garbage collection.
* **Old or Tenured Generation –** This is the part of Heap-memory that contains the older data objects that are not in frequent use or not in use at all are placed.
* **Permanent Generation –**This is the portion of Heap-memory that contains the JVM’s metadata for the runtime classes and application methods.

**Key Points:**

* We receive the corresponding error message if Heap-space is entirely full,  java. lang.OutOfMemoryError by JVM.
* This memory allocation scheme is different from the Stack-space allocation, here no automatic deallocation future is provided. We need to use a Garbage collector to remove the old unused objects in order to use the memory efficiently.
* The processing time(Accessing time) of this memory is quite slow as compared to Stack-memory.
* Heap-memory is also not threaded-safe as Stack-memory because data stored in Heap-memory are visible to all threads.
* Size of Heap-memory is quite larger as compared to the Stack-memory.
* Heap-memory is accessible or exists as long as the whole application(or java program) runs.

int main()

{

   // This memory for 10 integers

   // is allocated on heap.

   int \*ptr  = new int[10];

}

## Type conversions in c++

1. **Implicit Type Conversion** Also known as ‘automatic type conversion’.

* Done by the compiler on its own, without any external trigger from the user.
* Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid lose of data.
* All the data types of the variables are upgraded to the data type of the variable with largest data type.

bool -> char -> short int -> int ->

unsigned int -> long -> unsigned ->

long long -> float -> double -> long double

* It is possible for implicit conversions to lose information, signs can be lost (when signed is implicitly converted to unsigned), and overflow can occur (when long long is implicitly converted to float).

#include <iostream>

 int main(int argc, char const \*argv[])

 {

     int x = 10;

     char c = 'z';

     x = x + c;

     double var1 = 2348954;

     float var2 = var1 + 23.2;

     std::cout << "x = " << x << std::endl;

     std::cout << "c = " << c << std::endl;

     std::cout << "var2 = " << var2 << std::endl;

     return 0;

 }

# Note from Caleb’s lectures

* 1. Arrays are statically sized
  2. Vectors have dynamic size
  3. STL arrays are passed by value
  4. STL arrays are statically sized
  5. STL arrays are arrays wrapped inside the object
  6. Assigning to other variables is complicated C-like arrays, while templatized arrays and vectors can be assigned to other vars of respective types.
  7. Most of the times stick to the vectors

## Input/Output

Input stream doesn’t actually go to program one by one, but there is a buffer. It is an intermediate data storage where input is temporarily stored into memory.