Basics of C++

sstream

The standard header <sstream> defines a type called stringstream that allows a string to be treated as a stream, and thus allowing extraction or insertion operations from/to strings in the same way as they are performed on cin and cout. This feature is most useful to convert strings to numerical values and vice versa. For example, in order to extract an integer from a string we can write:

```
string mystr ("1204");
int myint;
stringstream(mystr) >>
myint;
```

This declares a string with initialized to a value of "1204", and a variable of type int. Then, the third line uses this variable to extract from a stringstream constructed from the string. This piece of code stores the numerical value 1204 in the variable called myint.

printf

```
#include <cstdio>

int main() {
    int num = 10;
    printf("The number is: %d\n", num);
    return 0;
}
```

scanf

```
#include <cstdio>
int main() {
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);
    printf("You entered: %d\n", num);
    return 0;
}
```

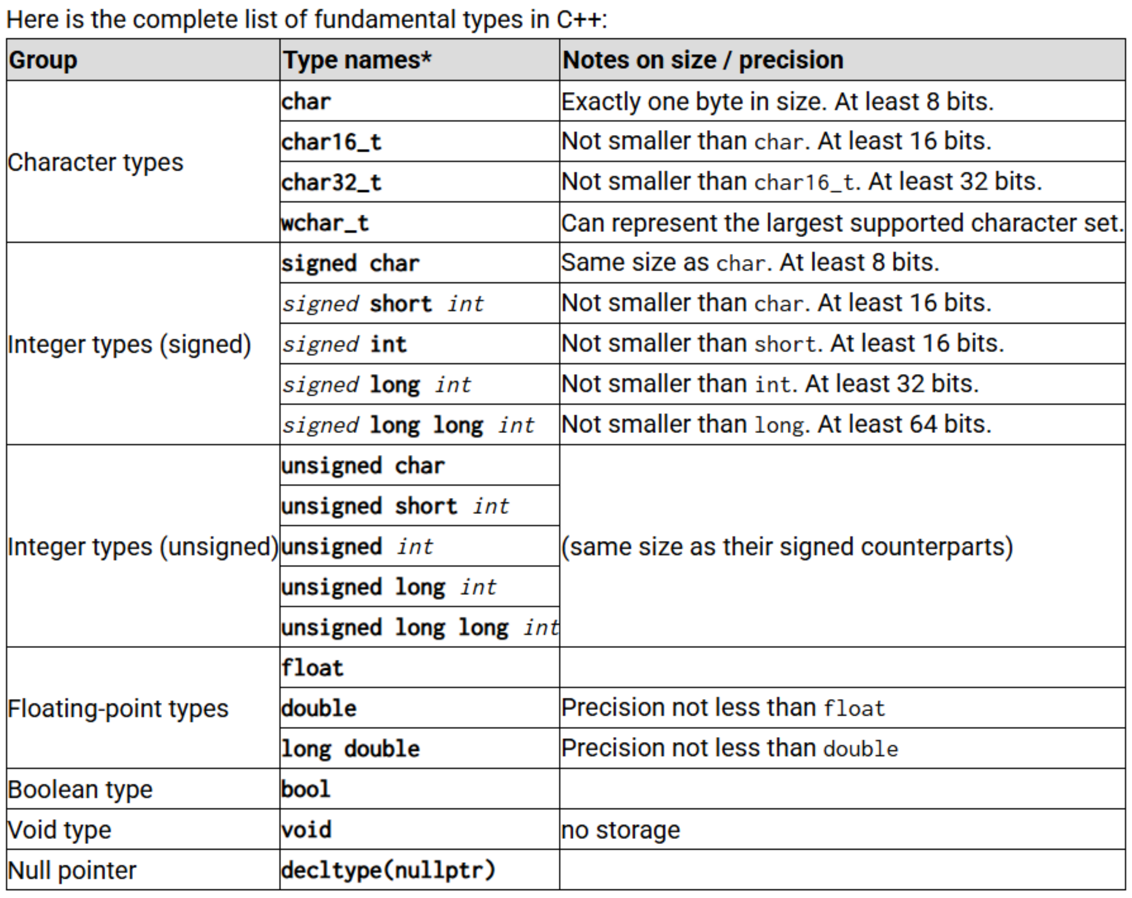
Compilers

Because a computer can only understand machine language and humans wish to write in high level languages. High level languages have to be re-written (translated) into machine language at some point. This is done by special programs called compilers, interpreters, or assemblers that are built into the various programming applications.

Identifiers

C++ uses a number of keywords to identify operations and data descriptions; therefore, identifiers created by a programmer cannot match these keywords.

Fundamental Data Types



Here is the complete list of fundamental types in C++:

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**	
Void type	**void**	no storage
Null pointer	**decltype(nullptr)**	

Initialization of variables

c-like initialization

```
type identifier = initial_value;
```

constructor initialization

```
type identifier (initial_value);
```

uniform initialization
`type identifier {initial_value};`

Type deduction: auto and decltype

```
int foo = 0;
auto bar = foo;  // the same as: int bar =
foo;
```

```
int foo = 0;
decltype(foo) bar;  // the same as: int
bar;
```

Constants

Literals

Three keyword literals exist in C++: `true`, `false` and `nullptr`:

- `true` and `false` are the two possible values for variables of type `bool`.
- `nullptr` is the *null pointer* value.

```
bool foo = true;
bool bar = false;
int* p = nullptr;
```

Typed constant expressions

```
const double pi = 3.1415926;
const char tab = '\t';
```

Another example:

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

const double pi = 3.14159;
const char newline = '\n';

int main ()
{
  double r=5.0;                // radius
  double circle;

  circle = 2 * pi * r;
  cout << circle;
  cout << newline;
}
```

Preprocessor Definitions (Another way)

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

```
#define identifier replacement

For example:

#define PI 3.14159
#define NEWLINE '\n'
```

Operators

- Assignment operator
- Arithmetic operator
- Compound assignment
- Increment and decrement
- Relational and comparison operator
- Logical operator
- Conditional ternary operator
- Comma operator

The comma operator (`,`) is used to separate two or more expressions that are included where only one expression is expected. When the set of expressions has to be evaluated for a value, only the right-most expression is considered.

For example, the following code:

```
a = (b=3, b+2);
```

would first assign the value 3 to `b`, and then assign `b+2` to variable `a`. So, at the end, variable `a` would contain the value 5 while variable `b` would contain value 3.

- Bitwise operator

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 to convert 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 (()):

```
int i;
float f = 3.14;
i = (int) f;
```

The previous code converts the floating-point number `3.14` to an integer value (`3`); the remainder is lost. Here, the typecasting operator was `(int)`. Another way to do the same thing in C++ is to use the functional notation preceding the expression to be converted by the type and enclosing the expression between parentheses:

```
i = int (f);
```

Both ways of casting types are valid in C++.

- Sizeof

This operator accepts one parameter, which can be either a type or a variable, and returns the size in bytes of that type or object:

```
x = sizeof (char);
```

Program Structure

Statements and Flow control

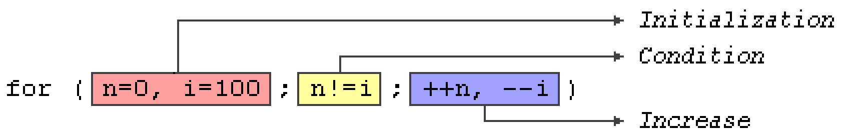
- Selection statement
  - If else
  - Switch (limited to constant expressions)

```
switch (expression)

{

  case constant1:

     group-of-statements-1;
```

```
    break;

  case constant2:

    group-of-statements-2;

    break;

  .
  .
  .

  default:

    default-group-of-statements

}
```

- Iteration statements
  - While
  - Do while
  - For loop

```
for (initialization; condition; increase) statement;
```



  - Range based loop

```
for ( declaration : range ) statement;
```

- Jump statements
  - Break
  - Continue
  - Goto

```cpp
// 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";
}
```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
liftoff!
```

`goto` is generally deemed a low-level feature, with no particular use cases in modern higher-level programming paradigms generally used with C++.

The destination point is identified by a *label*, which is then used as an argument for the `goto` statement. A *label* is made of a valid identifier followed by a colon (`:`).

Templates

Function templates

```cpp
template <class SomeType>
SomeType sum (SomeType a, SomeType b)
{
  return a+b;
}
```

`name <template-arguments> (function-arguments)`
For example, the `sum` function template defined above can be called with:

```cpp
x =
sum<int>(10,20);
```

The function `sum<int>` is just one of the possible instantiations of function template `sum`. In this case, by using `int` as template argument in the call, the compiler automatically instantiates a version of `sum` where each occurrence of `SomeType` is replaced by `int`, as if it was defined as:

```
int sum (int a, int b)
{
  return a+b;
}
```

```
// function template
#include <iostream>
using namespace std;

template <class T>
T sum (T a, T b)
{
  T result;
  result = a + b;
  return result;
}

int main () {
  int i=5, j=6, k;
  double f=2.0, g=0.5, h;
  k=sum<int>(i,j);
  h=sum<double>(f,g);
  cout << k << '\n';
  cout << h << '\n';
  return 0;
}
```

```
11
2.5
```

```
// function templates
#include <iostream>
using namespace std;

template <class T, class U>
bool are_equal (T a, U b)
{
  return (a==b);
}

int main ()
{
  if (are_equal(10,10.0))
    cout << "x and y are equal\n";
  else
    cout << "x and y are not equal\n";
  return 0;
}
```

Note that this example uses automatic template parameter deduction in the call to `are_equal`:

```
are_equal(10,10.0)
```

Is equivalent to:

```
are_equal<int,double>(10,10.0)
```

Name Visibility

Scopes

```
int some_function ()
{
  int x;
  x = 0;
  double x;    // wrong: name already used in this
scope
  x = 0.0;
}
```

```
// inner block scopes
#include <iostream>
using namespace std;

int main () {
  int x = 10;
  int y = 20;
  {
    int x;   // ok, inner
scope.
    x = 50;  // sets value to
inner x
    y = 50;  // sets value to
(outer) y
    cout << "inner block:\n";
    cout << "x: " << x << '\n';
    cout << "y: " << y << '\n';
  }
  cout << "outer block:\n";
  cout << "x: " << x << '\n';
  cout << "y: " << y << '\n';
  return 0;
}
```

```
inner block:
x: 50
y: 50
outer block:
x: 10
y: 50
```

Namespaces

Only one entity can exist with a particular name in a particular scope. This is seldom a problem for local names, since blocks tend to be relatively short, and names have particular purposes within them, such as naming a counter variable, an argument, etc…

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.

The syntax to declare a namespaces is:

```

namespace identifier
{
  named_entities
}

```

Where `identifier` is any valid identifier and `named_entities` is the set of variables, types and functions that are included within the namespace. For example:

```
namespace myNamespace
{
  int a, b;
}
```

In this case, the variables `a` and `b` are normal variables declared within a namespace called `myNamespace`.

These variables can be accessed from within their namespace normally, with their identifier (either `a` or `b`), but if accessed from outside the `myNamespace` namespace they have to be properly qualified with the scope operator `::`. For example, to access the previous variables from outside `myNamespace` they should be qualified like:

```
myNamespace::a
myNamespace::b
```

Namespaces are particularly useful to avoid name collisions. For example:

```
// namespaces
#include <iostream>
using namespace std;

namespace foo
{
  int value() { return 5; }
}

namespace bar
{
  const double pi = 3.1416;
  double value() { return 2*pi; }
}

int main () {
  cout << foo::value() << '\n';
  cout << bar::value() << '\n';
  cout << bar::pi << '\n';
  return 0;
}
```

```
5
6.2832
3.1416
```

using

The keyword `using` introduces a name into the current declarative region (such as a block), thus avoiding the need to qualify the name. For example:

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

namespace first
{
  int x = 5;
  int y = 10;
}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using first::x;
  using second::y;
  cout << x << '\n';
  cout << y << '\n';
  cout << first::y << '\n';
  cout << second::x << '\n';
  return 0;
}
```

```
5
2.7183
10
3.1416
```

The keyword `using` can also be used as a directive to introduce an entire namespace:

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

namespace first
{
  int x = 5;
  int y = 10;
}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using namespace first;
  cout << x << '\n';
  cout << y << '\n';
  cout << second::x << '\n';
  cout << second::y << '\n';
  return 0;
}
```

```
5
10
3.1416
2.7183
```

Compound Data Types

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.

A typical declaration for an array in C++ is:

```
type name [elements];
```

```cpp
int foo [5];
```

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, before the program runs.

Initializing:

```
int foo [5] = { 16, 2, 77, 40, 12071 };
```

```
int bar [5] = { 10, 20, 30 };
```

```
int baz [5] = { };
```

```
int foo [] = { 16, 2, 77, 40, 12071 };
```

```
int foo[] = { 10, 20, 30 };
int foo[] { 10, 20, 30 };
```

Character sequences

Initializing:

```
char myword[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
char myword[] = "Hello";
```

In any case, null-terminated character sequences and strings are easily transformed from one another:

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`:

```
char myntcs[] = "some text";
string mystring = myntcs;  // convert c-string to
string
cout << mystring;          // printed as a library
string
cout << mystring.c_str();  // printed as a c-string
```

(note: both `c_str` and `data` members of `string` are equivalent)

Pointers

The reference and dereference operators are thus complementary:

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

Pointers and arrays

The concept of arrays is related to that of pointers. In fact, 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. For example, consider these two declarations:

```
int myarray [20];
int * mypointer;
```

The following assignment operation would be valid:

```
mypointer = myarray;
```

After that, `mypointer` and `myarray` would be equivalent and would have very similar properties. The main difference being that `mypointer` can be assigned a different address, whereas `myarray` can never be assigned anything, and will always represent the same block of 20 elements of type `int`.

```
a[5] = 0;        // a [offset of 5] = 0
*(a+5) = 0;      // pointed to by (a+5) = 0
```

These two expressions are equivalent and valid, not only if `a` is a pointer, but also if `a` is an array. Remember that if an array, its name can be used just like a pointer to its first element.

Void pointers

The `void` type of pointer is a special type of pointer. In C++, `void` represents the absence of type. Therefore, `void` pointers are pointers that point to a value that has no type (and thus also an undetermined length and undetermined dereferencing properties).

This gives `void` pointers a great flexibility, by being able to point to any data type, from an integer value or a float to a string of characters. In exchange, they have a great limitation: 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 that points to a concrete data type before being dereferenced.

One of its possible uses may be to pass generic parameters to a function. For example:

```
// increaser
#include <iostream>
using namespace std;

void increase (void* data, int psize)
{
  if ( psize == sizeof(char) )
  { char* pchar; pchar=(char*)data; ++(*pchar);
}
  else if (psize == sizeof(int) )
  { int* pint; pint=(int*)data; ++(*pint); }
}

int main ()
{
  char a = 'x';
  int b = 1602;
  increase (&a,sizeof(a));
  increase (&b,sizeof(b));
  cout << a << ", " << b << '\n';
  return 0;
}
```

```
y, 1603
```

Invalid pointers

In principle, pointers are meant to point to valid addresses, such as the address of a variable or the address of an element in an array. But pointers can actually point to any address, including addresses that do not refer to any valid element. Typical examples of this are *uninitialized pointers* and pointers to nonexistent elements of an array:

```
int * p;               // uninitialized pointer (local
variable)

int myarray[10];
int * q = myarray+20;  // element out of bounds
```

Neither p nor q point to addresses known to contain a value, but none of the above statements causes an error. In C++, pointers are allowed to take any address value, no matter whether there actually is something at that address or not. What can cause an error is to dereference such a pointer (i.e., actually accessing the value they point to). Accessing such a pointer causes undefined behavior, ranging from an error during runtime to accessing some random value.

null pointers

But, sometimes, 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*. This value can be expressed in C++ in two ways: either with an integer value of zero, or with the `nullptr` keyword:

```
int * p = 0;
int * q = nullptr;
```

This also works (was used in older code)

```
int * r = NULL;
```

`NULL` is defined in several headers of the standard library, and is defined as an alias of some *null pointer* constant value (such as `0` or `nullptr`).

Pointers to function

```
// pointer to functions
#include <iostream>
using namespace std;

int addition (int a, int b)
{ return (a+b); }

int subtraction (int a, int b)
{ return (a-b); }

int operation (int x, int y, int (*functocall)(int,int))
{
  int g;
  g = (*functocall)(x,y);
  return (g);
}

int main ()
{
  int m,n;
  int (*minus)(int,int) = subtraction;

  m = operation (7, 5, addition);
  n = operation (20, m, minus);
  cout <<n;
  return 0;
}
```

```
8
```

Dynamic Memory

programs need to dynamically allocate memory, for which the C++ language integrates the operators `new` and `delete`.

Operators new and new[]

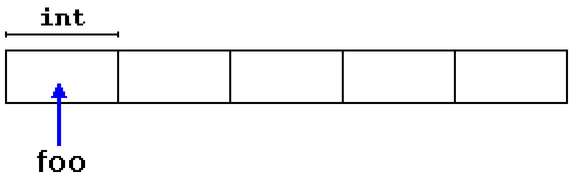
Dynamic memory is allocated using operator `new`. `new` is followed by a data type specifier and, if a sequence of more than one element is required, the number of these within brackets `[]`. It returns a pointer to the beginning of the new block of memory allocated. Its syntax is:

```
pointer = new type
pointer = new type [number_of_elements]
```

The first expression is used to allocate memory to contain one single element of type `type`. The second one is used to allocate a block (an array) of elements of type `type`, where `number_of_elements` is an integer value representing the amount of these. For example:

```
int * foo;
foo = new int [5];
```

In this case, the system dynamically allocates space for five elements of type `int` and returns a pointer to the first element of the sequence, which is assigned to `foo` (a pointer). Therefore, `foo` now points to a valid block of memory with space for five elements of type `int`.



Here, `foo` is a pointer, and thus, the first element pointed to by `foo` can be accessed either with the expression `foo[0]` or the expression `*foo` (both are equivalent). The second element can be accessed either with `foo[1]` or `*(foo+1)`, and so on...

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.

Handling unsuccessful allocation:

The other method is known as `nothrow`, and 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.

This method can be specified by using a special object called `nothrow`, declared in header [`<new>`](), as argument for `new`:

```
foo = new (nothrow) int
[5];
```

In this case, if the allocation of this block of memory fails, the failure can be detected by checking if `foo` is a null pointer:

```
int * foo;
foo = new (nothrow) int [5];
if (foo == nullptr) {
  // error assigning memory. Take
measures.
}
```

Operator delete and delete[]

In most cases, memory allocated dynamically is only needed during specific periods of time within a program; once it is no longer needed, it can be freed so that the memory becomes available again for other requests of dynamic memory. This is the purpose of operator `delete`, whose syntax is:

```
delete pointer;
delete[] pointer;
```

```cpp
// rememb-o-matic
#include <iostream>
#include <new>
using namespace std;

int main ()
{
  int i,n;
  int * p;
  cout << "How many numbers would you like to type? ";
  cin >> i;
  p= new (nothrow) int[i];
  if (p == nullptr)
    cout << "Error: memory could not be allocated";
  else
  {
    for (n=0; n<i; n++)
    {
      cout << "Enter number: ";
      cin >> p[n];
    }
    cout << "You have entered: ";
    for (n=0; n<i; n++)
      cout << p[n] << ", ";
    delete[] p;
  }
  return 0;
}
```

```
How many numbers would you like to type? 5
Enter number : 75
Enter number : 436
Enter number : 1067
Enter number : 8
Enter number : 32
You have entered: 75, 436, 1067, 8, 32,
```

Data Structures

A `data structure` is a group of data elements grouped together under one name. These data elements, known as *members*, can have different types and different lengths. Data structures can be declared in C++ using the following syntax:

```
struct type_name {
member_type1 member_name1;
member_type2 member_name2;
member_type3 member_name3;
.
.
} object_names;
```

```cpp
struct product {
  int weight;
  double price;
} ;

product apple;
product banana, melon;
```

```cpp
struct product {
  int weight;
  double price;
} apple, banana, melon;
```

```cpp
// pointers to structures
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

struct movies_t {
  string title;
  int year;
};

int main ()
{
  string mystr;

  movies_t amovie;
  movies_t * pmovie;
  pmovie = &amovie;

  cout << "Enter title: ";
  getline (cin, pmovie->title);
  cout << "Enter year: ";
  getline (cin, mystr);
  (stringstream) mystr >> pmovie->year;

  cout << "\nYou have entered:\n";
  cout << pmovie->title;
  cout << " (" << pmovie->year << ")\n";

  return 0;
```

```
Enter title:
Invasion of the
body snatchers
Enter year: 1978

You have
entered:Invasion
of the body
snatchers (1978)
```

```
}
```

Pointers to structures

Like any other type, structures can be pointed to by its own type of pointers:

```
struct movies_t {
  string title;
  int year;
};

movies_t amovie;
movies_t * pmovie;
```

Here `amovie` is an object of structure type `movies_t`, and `pmovie` is a pointer to point to objects of structure type `movies_t`. Therefore, the following code would also be valid:

```
pmovie = &amovie;
```

The arrow operator (`->`) is a dereference operator that is used exclusively with pointers to objects that have members. This operator serves to access the member of an object directly from its address. For example, in the example above:

```
pmovie->
```

is, for all purposes, equivalent to:

```
(*pmovie).
```

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)`

Other Data Types

Type aliases (typedef / using)

A type alias is a different name by which a type can be identified. In C++, any valid type can be aliased so that it can be referred to with a different identifier.

In C++, there are two syntaxes for creating such type aliases: The first, inherited from the C language, uses the `typedef` keyword:

```
typedef existing_type new_type_name ;
```

where `existing_type` is any type, either fundamental or compound, and `new_type_name` is an identifier with the new name given to the type.

For example:

```
typedef char C;
typedef unsigned int WORD;
typedef char * pChar;
typedef char field [50];
```

This defines four type aliases: `C`, `WORD`, `pChar`, and `field` as `char`, `unsigned int`, `char*` and `char[50]`, respectively. Once these aliases are defined, they can be used in any declaration just like any other valid type:

```
C mychar, anotherchar, *ptc1;
WORD myword;
pChar ptc2;
field name;
```

More recently, a second syntax to define type aliases was introduced in the C++ language:

```
using new_type_name = existing_type ;
```

For example, the same type aliases as above could be defined as:

```
using C = char;
using WORD = unsigned int;
using pChar = char *;
using field = char [50];
```

Unions

Unions allow one portion of memory to be accessed as different data types. Its declaration and use is similar to the one of structures, but its functionality is totally different:

```
union type_name {
  member_type1 member_name1;
  member_type2 member_name2;
  member_type3 member_name3;
  .
  .
} object_names;
```

I don’t think I will need this thing?

Enumerated Types (Enum)

```
enum type_name {
  value1,
  value2,
  value3,
  .
  .
} object_names;
```

Enumerated types with enum class

But, in C++, 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`:

```
enum class Colors {black, blue, green, cyan, red, purple, yellow, white};
```

Each of the enumerator values of an `enum class` type needs to be scoped into its type (this is actually also possible with `enum` types, but it is only optional). For example:

```
Colors mycolor;

mycolor = Colors::blue;
if (mycolor == Colors::green) mycolor = Colors::red;
```

Enumerated types declared with `enum class` also have more control over their underlying type; it may be any integral data type, such as `char`, `short` or `unsigned int`, which essentially serves to determine the size of the type. This is specified by a colon and the underlying type following the enumerated type. For example:

```
enum class EyeColor : char {blue, green, brown};
```

Here, `Eyecolor` is a distinct type with the same size of a `char` (1 byte).

Classes

Classes are defined using either keyword `class` or keyword `struct`, with the following syntax:

```
class class_name {
  access_specifier_1:
    member1;
  access_specifier_2:
    member2;
  ...
} object_names;
```

Where `class_name` is a valid identifier for the class, `object_names` is an optional list of names for objects of this class. The body of the declaration can contain *members*, which can either be data or function declarations, and optionally *access specifiers*.

Classes have the same format as plain *data structures*, **except that they can also include functions and have these new things called *access specifiers***. An *access specifier* is one of the following three keywords: `private`, `public` or `protected`.

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.

```
// classes example
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    void set_values (int,int);
    int area() {return width*height;}
};

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

int main () {
  Rectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area();
  return 0;
}
```

```
area: 12
```

Constructors

What would happen in the previous example if we called the member function `area` before having called `set_values`? An undetermined result, since the members `width` and `height` had never been assigned a value.

In order to avoid that, 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**.

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`.

Overloading Constructors

```
// overloading class constructors
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    Rectangle ();
    Rectangle (int,int);
    int area (void) {return
(width*height);}
};

Rectangle::Rectangle () {
  width = 5;
  height = 5;
}

Rectangle::Rectangle (int a, int b) {
  width = a;
  height = b;
}

int main () {
  Rectangle rect (3,4);
  Rectangle rectb;
  cout << "rect area: " << rect.area() <<
endl;
  cout << "rectb area: " << rectb.area() <<
endl;
  return 0;
}
```

```
rect area: 12
rectb area: 25
```

The ***default constructor*** is the constructor that takes no parameters, and it is special because it is called when an object is declared but is not initialized with any arguments.

Uniform Initialization

The way of calling constructors by enclosing their arguments in parentheses, as shown above, is known as *functional form*. But constructors can also be called with other syntaxes:

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, ... }
```

Ex:

```
// classes and uniform initialization
#include <iostream>
using namespace std;

class Circle {
    double radius;
  public:
    Circle(double r) { radius = r; }
    double circum() {return
2*radius*3.14159265;}
};

int main () {
  Circle foo (10.0);   // functional form
  Circle bar = 20.0;   // assignment init.
  Circle baz {30.0};   // uniform init.
  Circle qux = {40.0}; // POD-like

  cout << "foo's circumference: " <<
foo.circum() << '\n';
  return 0;
}
```

```
foo's
circumference:
62.8319
```

Member Initialisation Methods

```
class Rectangle {
    int width,height;
  public:
    Rectangle(int,int);
    int area() {return width*height;}
};
```

1. Usual method

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

2. Two other methods

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

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

Pointers to Class

```
// pointer to classes example
#include <iostream>
using namespace std;

class Rectangle {
  int width, height;
public:
  Rectangle(int x, int y) : width(x), height(y) {}
  int area(void) { return width * height; }
};


int main() {
  Rectangle obj (3, 4);
  Rectangle * foo, * bar, * baz;
  foo = &obj;
  bar = new Rectangle (5, 6);
  baz = new Rectangle[2] { {2,5}, {3,6} };
  cout << "obj's area: " << obj.area() << '\n';
  cout << "*foo's area: " << foo->area() << '\n';
  cout << "*bar's area: " << bar->area() << '\n';
  cout << "baz[0]'s area:" << baz[0].area() << '\n';
  cout << "baz[1]'s area:" << baz[1].area() << '\n';
  delete bar;
  delete[] baz;
  return 0;
}
```

Class vs Struct

Classes can be defined not only with keyword `class`, but also with keywords `struct` and `union`.

The keyword `struct`, generally used to declare plain data structures, can also be used to declare classes that have member functions, with the same syntax as with keyword `class`. The only difference between both is that members of classes declared with the keyword `struct` have `public` access by default, while members of classes declared with the keyword `class` have `private` access by default. For all other purposes both keywords are equivalent in this context.

~~Conversely, the concept of *unions* is different from that of classes declared with `struct` and `class`, since unions only store one data member at a time, but nevertheless they are also classes and can thus also hold member functions. The default access in union classes is `public`.~~

Overloading Operators

```
type operator sign (parameters) { /*... body ...*/ }
```

```
// overloading operators example
#include <iostream>
using namespace std;

class CVector {
  public:
    int x,y;
    CVector () {};
    CVector (int a,int b) : x(a), y(b) {}
    CVector operator + (const CVector&);
};

CVector CVector::operator+ (const CVector& param) {
  CVector temp;
  temp.x = x + param.x;
  temp.y = y + param.y;
  return temp;
}

int main () {
  CVector foo (3,1);
  CVector bar (1,2);
  CVector result;
  result = foo + bar;
  cout << result.x << ',' << result.y << '\n';
  return 0;
}
```

```
4,3
```

```
// non-member operator overloads
#include <iostream>
using namespace std;

class CVector {
  public:
    int x,y;
    CVector () {}
    CVector (int a, int b) : x(a), y(b) {}
};


CVector operator+ (const CVector& lhs, const CVector&
rhs) {
  CVector temp;
  temp.x = lhs.x + rhs.x;
  temp.y = lhs.y + rhs.y;
  return temp;
}

int main () {
  CVector foo (3,1);
  CVector bar (1,2);
  CVector result;
  result = foo + bar;
  cout << result.x << ',' << result.y << '\n';
  return 0;
}
```
	4,3

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.

One of its uses can be to check if a parameter passed to a member function is the object itself. For example:

```
// example on this
#include <iostream>
using namespace std;

class Dummy {
  public:
    bool isitme (Dummy& param);
};

bool Dummy::isitme (Dummy& param)
{
  if (&param == this) return true;
  else return false;
}

int main () {
  Dummy a;
  Dummy* b = &a;
  if ( b->isitme(a) )
    cout << "yes, &a is b\n";
  return 0;
}
```

```
yes, &a is b
```

Static Members

A class can contain static members, either data or functions.

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.

For example, it may be used for a variable within a class that can contain a counter with the number of objects of that class that are currently allocated, as in the following example:

```
// static members in classes
#include <iostream>
using namespace std;

class Dummy {
  public:
    static int n;
    Dummy () { n++; };
};

int Dummy::n=0;

int main () {
  Dummy a;
  Dummy b[5];
  cout << a.n << '\n';
  Dummy * c = new Dummy;
  cout << Dummy::n << '\n';
  delete c;
  return 0;
}
```

```
6
7
```

In fact, static members have the same properties as non-member variables but they enjoy class scope. For that reason, and to avoid them to be declared several times, they cannot be initialized directly in the class, but need to be initialized somewhere outside it. As in the previous example:

```
int Dummy::n=0;
```

Because it is a common variable value for all the objects of the same class, it can be referred to as a member of any object of that class or even directly by the class name (of course this is only valid for static members):

```
cout << a.n;
cout << Dummy::n;
```

These two calls above are referring to the same variable: the static variable `n` within class `Dummy` shared by all objects of this class.

Again, it is just like a non-member variable, but with a name that requires to be accessed like a member of a class (or an object).

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`.

Const member functions

When an object of a class is qualified as a `const` object:

```
const MyClass myobject;
```

The access to its 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:

```
// constructor on const object
#include <iostream>
using namespace std;

class MyClass {
  public:
    int x;
    MyClass(int val) : x(val) {}
    int get() {return x;}
};

int main() {
  const MyClass foo(10);
// foo.x = 20;            // not valid: x cannot be modified
  cout << foo.x << '\n';  // ok: data member x can be read
  return 0;
}
```

```
10
```

The member functions of a `const` object can only be called if they are themselves specified as `const` members; in the example above, member `get` (which is not specified as `const`) cannot be called from `foo`. To specify that a member is a `const` member, the `const` keyword shall follow the function prototype, after the closing parenthesis for its parameters:

```
int get() const {return x;}
```

Note that `const` can be used to qualify the type returned by a member function. This `const` is not the same as the one which specifies a member as `const`. Both are independent and are located at different places in the function prototype:

```
int get() const {return x;}        // const member function
const int& get() {return x;}       // member function returning a const&
const int& get() const {return x;} // const member function returning a const&
```

Member functions specified to be `const` cannot modify non-static data members nor call other non-`const` member functions. In essence, `const` members shall not modify the state of an object.

`const` objects are limited to access only member functions marked as `const`, but non-`const` objects are not restricted and thus can access both `const` and non-`const` member functions alike.

You may think that anyway you are seldom going to declare `const` objects, and thus marking all members that don't modify the object as const is not worth the effort, but const objects are actually very common. Most functions taking classes as parameters actually take them by `const` reference, and thus, these functions can only access their `const` members:

```cpp
// const objects
#include <iostream>
using namespace std;

class MyClass {
    int x;
  public:
    MyClass(int val) : x(val) {}
    const int& get() const {return x;}
};

void print (const MyClass& arg) {
  cout << arg.get() << '\n';
}

int main() {
  MyClass foo (10);
  print(foo);

  return 0;
}
```

```
10
```

If in this example, `get` was not specified as a `const` member, the call to `arg.get()` in the `print` function would not be possible, because `const` objects only have access to `const` member functions.

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`.

```cpp
// overloading members on constness
#include <iostream>
using namespace std;

class MyClass {
    int x;
  public:
    MyClass(int val) : x(val) {}
    const int& get() const {return x;}
    int& get() {return x;}
};

int main() {
  MyClass foo (10);
  const MyClass bar (20);
  foo.get() = 15;         // ok: get() returns int&
// bar.get() = 25;        // not valid: get() returns const int&
  cout << foo.get() << '\n';
  cout << bar.get() << '\n';

  return 0;
}
```

```
15
20
```

Class Templates

Just like we can create function templates, we can also create class templates, allowing classes to have members that use template parameters as types. For example:

```
template <class T>
class mypair {
    T values [2];
  public:
    mypair (T first, T second)
    {
      values[0]=first; values[1]=second;
    }
};
```

The class that we have just defined serves to store two elements of any valid type. For example, if we wanted to declare an object of this class to store two integer values of type `int` with the values 115 and 36 we would write:

```
mypair<int> myobject (115, 36);
```

This same class could also be used to create an object to store any other type, such as:

```
mypair<double> myfloats (3.0, 2.18);
```

The constructor is the only member function in the previous class template and it has been defined inline within the class definition itself. In case that a member function is defined outside the definition of the class template, it shall be preceded with the `template <...>` prefix:

```
// class templates
#include <iostream>
using namespace std;

template <class T>
class mypair {
    T a, b;
  public:
    mypair (T first, T second)
      {a=first; b=second;}
    T getmax ();
};

template <class T>
T mypair<T>::getmax ()
{
  T retval;
  retval = a>b? a : b;
  return retval;
}

int main () {
  mypair <int> myobject (100, 75);
  cout << myobject.getmax();
  return 0;
}
```

```
100
```

Notice the syntax of the definition of member function `getmax`:

```
template <class T>
T mypair<T>::getmax ()
```

Confused by so many `T`'s? There are three `T`'s in this declaration: The first one is the template parameter. The second `T` refers to the type returned by the function. And the third `T` (the one between angle brackets) is also a requirement: It specifies that this function's template parameter is also the class template parameter.

Template Specialisation

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***.

For example, let's suppose that we have a very simple class called `mycontainer` that can store one element of any type and that has just one member function called `increase`, which increases its value. But we find that when it stores an element of type `char` it would be more convenient to have a completely different implementation with a function member `uppercase`, so we decide to declare a class template specialization for that type:

```
// 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 ('j');
  cout << myint.increase() << endl;
  cout << mychar.uppercase() << endl;
  return 0;
}
```

```
8
J
```

This is the syntax used for the class template specialization:

```
template <> class mycontainer <char> { ... };
```

But more important than this prefix, is the `<char>` specialization parameter after the class template name. This specialization parameter itself identifies the type for which the template class is being specialized (`char`). Notice the differences between the generic class template and the specialization:

```
template <class T> class mycontainer { ... };
template <> class mycontainer <char> { ... };
```

The first line is the generic template, and the second one is the specialization.

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. There are six:

Member function	**typical form for class C:**
Default constructor	`C::C();`
Destructor	`C::~C();`
Copy constructor	`C::C (const C&);`
Copy assignment	`C& operator= (const C&);`
Move constructor	`C::C (C&&);`
Move assignment	`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.

—--

If a class definition has no constructors, the compiler assumes the class to have an implicitly defined *default constructor*. Therefore, after declaring a class like this:

```
class Example {
  public:
    int total;
    void accumulate (int x) { total += x; }
};
```

The compiler assumes that `Example` has a *default constructor*. Therefore, objects of this class can be constructed by simply declaring them without any arguments:

```
Example ex;
```

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. For example, the following class:

```
class Example2 {
  public:
    int total;
    Example2 (int initial_value) : total(initial_value) { };
    void accumulate (int x) { total += x; };
};
```

Here, we have declared a constructor with a parameter of type `int`. Therefore the following object declaration would be correct:

```
Example2 ex (100);    // ok: calls constructor
```

But the following:

```
Example2 ex;          // not valid: no default constructor
```

Would not be valid, since the class has been declared with an explicit constructor taking one argument and that replaces the implicit *default constructor* taking none.

Therefore, if objects of this class need to be constructed without arguments, the proper *default constructor* shall also be declared in the class. For example:

```
// classes and default constructors
#include <iostream>
#include <string>
using namespace std;

class Example3 {
    string data;
  public:
    Example3 (const string& str) : data(str) {}
    Example3() {}
    const string& content() const {return data;}
};

int main () {
  Example3 foo;
  Example3 bar ("Example");

  cout << "bar's content: " << bar.content() <<
'\n';
  return 0;
}
```

```
bar's
content:
Example
```

Here, `Example3` has a *default constructor* (i.e., a constructor without parameters) defined as an empty block:

```
Example3() {}
```

This allows objects of class `Example3` to be constructed without arguments (like `foo` was declared in this example). Normally, a default constructor like this is implicitly defined for all classes that have no other constructors and thus no explicit definition is required. But in this case, `Example3` has another constructor:

```
Example3 (const string& str);
```

And when any constructor is explicitly declared in a class, no implicit *default constructors* are 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. The classes we have defined in previous chapters did not allocate any resources and thus did not really require any clean up.

But now, let's imagine that the class in the last example allocates dynamic memory to store the string it had as data member; in this case, 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`. It also uses the class name as its own name, but preceded with a tilde sign (`~`):

```
// destructors
#include <iostream>
#include <string>
using namespace std;

class Example4 {
    string* ptr;
  public:
    // constructors:
    Example4() : ptr(new string) {}
    Example4 (const string& str) : ptr(new
string(str)) {}
    // destructor:
    ~Example4 () {delete ptr;}
    // access content:
    const string& content() const {return *ptr;}
};

int main () {
  Example4 foo;
  Example4 bar ("Example");

  cout << "bar's content: " << bar.content() <<
'\n';
  return 0;
}
```

```
bar's
content:
Example
```

On construction, `Example4` allocates storage for a `string`. Storage that is later released by the destructor.

The destructor for an object is called at the end of its lifetime; in the case of `foo` and `bar` this happens at the end of function `main`.

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. For example, for a class `MyClass`, the *copy constructor* may have the following signature:

```
MyClass::MyClass (const MyClass&);
```

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) {}
```

This default *copy constructor* may suit the needs of many classes. But *shallow copies* only copy the members of the class themselves, and this is probably not what we expect for classes like class `Example4` we defined above, because it contains pointers of which it handles its storage. For that class, performing a *shallow copy* means that the pointer value is copied, but not the content itself; **This means that both objects (the copy and the original) would be sharing a single `string` object (they would both be pointing to the same object), and at some point (on destruction) both objects would try to delete the same block of memory, probably causing the program to crash on runtime. This can be solved by defining the following custom *copy constructor* that performs a *deep copy*:**

```
// copy constructor: deep copy
#include <iostream>
#include <string>
using namespace std;

class Example5 {
    string* ptr;
  public:
    Example5 (const string& str) : ptr(new
string(str)) {}
    ~Example5 () {delete ptr;}
    // copy constructor:
    Example5 (const Example5& x) : ptr(new
string(x.content())) {}
    // access content:
    const string& content() const {return *ptr;}
};

int main () {
  Example5 foo ("Example");
  Example5 bar = foo;

  cout << "bar's content: " << bar.content() <<
'\n';
  return 0;
}
```

```
bar's
content:
Example
```

The *deep copy* performed by this *copy constructor* allocates storage for a new string, which is initialized to contain a copy of the original object. In this way, both objects (copy and original) have distinct copies of the content stored in different locations.

Copy assignment

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

```
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.

The *copy assignment operator* is an overload of `operator=` which takes a *value* or *reference* of the class itself as a 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, as is the case in `Example5`. 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*:

```
Example5& operator= (const Example5& 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:

```
Example5& operator= (const Example5& 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. These cases trigger the ***move constructor*** **and** ***move assignments*****:**

The *move constructor* is called when an object is initialized on construction using an unnamed temporary. Likewise, the *move assignment* is called when an object is assigned the value of an unnamed temporary:

```
MyClass fn();            // function returning a MyClass object
MyClass foo;             // default constructor
MyClass bar = foo;       // copy constructor
MyClass baz = fn();      // move constructor
foo = bar;               // copy assignment
baz = MyClass();         // move assignment
```

Both the value returned by `fn` and the value constructed with `MyClass` are unnamed temporaries. In these cases, there is no need to make a copy, because the unnamed object is very short-lived and can be acquired by the other object when this is a more efficient operation.

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.

```cpp
// move constructor/assignment
#include <iostream>
#include <string>
using namespace std;

class Example6 {
    string* ptr;
  public:
    Example6 (const string& str) : ptr(new
string(str)) {}
    ~Example6 () {delete ptr;}
    // move constructor
    Example6 (Example6&& x) : ptr(x.ptr)
{x.ptr=nullptr;}
    // move assignment
    Example6& operator= (Example6&& x) {
      delete ptr;
      ptr = x.ptr;
      x.ptr=nullptr;
      return *this;
    }
    // access content:
    const string& content() const {return *ptr;}
    // addition:
    Example6 operator+(const Example6& rhs) {
      return Example6(content()+rhs.content());
    }
};

int main () {
  Example6 foo ("Exam");
  Example6 bar = Example6("ple");   //
move-construction

  foo = foo + bar;                  //
move-assignment

  cout << "foo's content: " << foo.content() <<
'\n';
  return 0;
}
```

```
foo's
content:
Example
```

Friendship and Inheritance

Friend functions

In principle, private and protected members of a class cannot be accessed from outside the same class in which they are declared. However, this rule does not apply to *"friends"*.

`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`:

```cpp
// friend functions
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    Rectangle() {}
    Rectangle (int x, int y) : width(x), height(y) {}
    int area() {return width * height;}
    friend Rectangle duplicate (const Rectangle&);
};

Rectangle duplicate (const Rectangle& param)
{
  Rectangle res;
  res.width = param.width*2;
  res.height = param.height*2;
  return res;
}

int main () {
  Rectangle foo;
  Rectangle bar (2,3);
  foo = duplicate (bar);
  cout << foo.area() << '\n';
  return 0;
}
```

```
24
```

Notice that neither in the declaration of `duplicate` nor in its later use in `main`, function `duplicate` is considered a member of class `Rectangle`. It isn't! It simply has access to its private and protected members without being a member.

Friend classes

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

```cpp
// friend class
#include <iostream>
using namespace std;

class Square;

class Rectangle {
    int width, height;
  public:
    int area ()
      {return (width * height);}
    void convert (Square a);
};

class Square {
  friend class Rectangle;
  private:
    int side;
  public:
    Square (int a) : side(a) {}
};

void Rectangle::convert (Square a) {
  width = a.side;
  height = a.side;
}

int main () {
  Rectangle rect;
  Square sqr (4);
  rect.convert(sqr);
  cout << rect.area();
  return 0;
}
```

```
16
```

Inheritance between classes

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.

```
class derived_class_name: public base_class_name
{ /*...*/ };
```

(The `public` access specifier may be replaced by any one of the other access specifiers (`protected` or `private`). 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.)

1. Public inheritance
   - Public members of the base class remain public in the derived class.
   - Protected members of the base class remain protected in the derived class.
   - Private members of the base class remain inaccessible in the derived class (private members are never inherited directly).

2. Protected inheritance
   - Public members of the base class become protected in the derived class.
   - Protected members of the base class remain protected in the derived class.
   - Private members of the base class remain inaccessible in the derived class.

3. Private inheritance
   - Public members of the base class become private in the derived class.
   - Protected members of the base class become private in the derived class.
   - Private members of the base class remain inaccessible in the derived class.

```
// derived classes
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
 };

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;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  cout << rect.area() << '\n';
  cout << trgl.area() << '\n';
  return 0;
}
```

```
20
10
```

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`.

Multiple Inheritance

A class may inherit from more than one class by simply specifying more base classes, separated by commas, in the list of a class's base classes (i.e., after the colon). For example, if the program had a specific class to print on screen called `Output`, and we wanted our classes `Rectangle` and `Triangle` to also inherit its members in addition to those of `Polygon` we could write:

```
class Rectangle: public Polygon, public Output;
class Triangle: public Polygon, public Output;
```

```
// multiple inheritance
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    Polygon (int a, int b) : width(a), height(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 (int a, int b) : Polygon(a,b) {}
    int area ()
      { return width*height; }
};

class Triangle: public Polygon, public Output {
  public:
    Triangle (int a, int b) : Polygon(a,b) {}
    int area ()
      { return width*height/2; }
};

int main () {
  Rectangle rect (4,5);
  Triangle trgl (4,5);
  rect.print (rect.area());
  Triangle::print (trgl.area());
  return 0;
}
```

```
20
10
```

Polymorphism

Pointers to base class

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. *Polymorphism* is the art of taking advantage of this simple but powerful and versatile feature.

```cpp
// pointers to base class
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
};

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 * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << rect.area() << '\n';
  cout << trgl.area() << '\n';
  return 0;
}
```

```
20
10
```

For example, the following two statements would be equivalent in the previous example:

```
ppoly1->set_values (4,5);
rect.set_values (4,5);
```

Virtual Members

A virtual member is a member function that can be redefined in a derived class, while preserving its calling properties through references. The syntax for a function to become virtual is to precede its declaration with the `virtual` keyword:

```cpp
// 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;
}
```

```
20
10
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.**

Therefore, **essentially, 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*.**

Abstract Base Classes

Abstract base classes are something very similar to the `Polygon` class in the previous example. **They are classes that can only be used as base classes, and thus are allowed to have virtual member functions without definition (known as pure virtual functions)**.

The syntax is to **replace their definition by `=0`** (an equal sign and a zero):

An abstract base `Polygon` class could look like this:

```
// abstract class CPolygon
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area () =0;
};
```

- **Classes that contain at least one *pure virtual function* are known as *abstract base classes*.**
- **Abstract base classes cannot be used to instantiate objects.**
- But an *abstract base class* is not totally useless. **It can be used to create pointers to it, and take advantage of all its polymorphic abilities.**

```
Polygon mypolygon;   // not working if Polygon is abstract base class
```

This works:

```
Polygon * ppoly1;
Polygon * ppoly2;
```

And can actually be dereferenced when pointing to objects of derived (non-abstract) classes. Here is the entire example:

```
// abstract base class
#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 (void) =0;
};

class Rectangle: public Polygon {
  public:
    int area (void)
      { return (width * height); }
};

class Triangle: public Polygon {
  public:
    int area (void)
      { return (width * height / 2); }
};

int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << ppoly1->area() << '\n';
  cout << ppoly2->area() << '\n';
  return 0;
}
```

```
20
10
```

In this example, objects of different but related types are referred to using a unique type of pointer (`Polygon*`) and the proper member function is called every time, just because they are virtual. This can be really useful in some circumstances. For example, it is even possible for a member of the abstract base class `Polygon` **to use the special pointer `this` to access the proper virtual members**, even though `Polygon` itself has no implementation for this function:

```
// pure virtual members can be called
// from the abstract base class
#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() =0;
    void printarea()
      { cout << this->area() << '\n'; }
};

class Rectangle: public Polygon {
  public:
    int area (void)
      { return (width * height); }
};

class Triangle: public Polygon {
  public:
    int area (void)
      { return (width * height / 2); }
};

int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly1->printarea();
  ppoly2->printarea();
```

```
20
10
```

```
  return 0;
}
```

Just another example combining some things we learnt before- dynamic memory, constructor initializers and polymorphism:

```
// dynamic allocation and polymorphism
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    Polygon (int a, int b) : width(a), height(b) {}
    virtual int area (void) =0;
    void printarea()
      { cout << this->area() << '\n'; }
};

class Rectangle: public Polygon {
  public:
    Rectangle(int a,int b) : Polygon(a,b) {}
    int area()
      { return width*height; }
};

class Triangle: public Polygon {
  public:
    Triangle(int a,int b) : Polygon(a,b) {}
    int area()
      { return width*height/2; }
};

int main () {
  Polygon * ppoly1 = new Rectangle (4,5);
  Polygon * ppoly2 = new Triangle (4,5);
  ppoly1->printarea();
  ppoly2->printarea();
  delete ppoly1;
  delete ppoly2;
  return 0;
}
```

```
20
10
```

Other Features

Type Conversions

Implicit conversion

Implicit conversions are **automatically performed when a value is copied to a compatible type**. For example:

```
short a=2000;
int b;
b=a;
```

Here, the value of `a` is promoted from `short` to `int` **without the need of any explicit operator. This is known as a *standard conversion*. 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.**

- 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).

- 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.

Implicit conversions with classes

In the world of classes, implicit conversions can be controlled by means of three member functions:

- **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;
}
```

Keyword explicit

Skipped

Type casting

There exist two main syntaxes for generic type-casting: *functional* and *c-like*:

```
double x = 10.3;
int y;
y = int (x);    // functional notation
y = (int) x;    // c-like cast notation
```

```
dynamic_cast <new_type> (expression)
reinterpret_cast <new_type> (expression)
static_cast <new_type> (expression)
const_cast <new_type> (expression)
```

Details about these are skipped.

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

```
(new_type) expression
new_type (expression)
```

Typeid

`typeid` allows to check the type of an expression:

```
typeid (expression)
```

This operator returns a reference to a constant object of type `type_info` that is defined in the standard header `<typeinfo>`. A value returned by `typeid` can be compared with another value returned by `typeid` using operators `==` and `!=` or can serve to obtain a null-terminated character sequence representing the data type or class name by using its **`name()` member**.

```
// typeid
#include <iostream>
#include <typeinfo>
using namespace std;

int main () {
  int * a,b;
  a=0; b=0;
  if (typeid(a) != typeid(b))
  {
    cout << "a and b are of different types:\n";
    cout << "a is: " << typeid(a).name() << '\n';
    cout << "b is: " << typeid(b).name() << '\n';
  }
  return 0;
}
```

```
a and b
are of
different
types:
a is: int
*
b is: int
```

When `typeid` is applied to classes, `typeid` uses the RTTI (*Run-Time Type Information)* to keep track of the type of dynamic objects. When `typeid` is applied to an expression whose type is a polymorphic class, the result is the type of the most derived complete object:

```
// typeid, polymorphic class
#include <iostream>
#include <typeinfo>
#include <exception>
using namespace std;

class Base { virtual void f(){} };
class Derived : public Base {};

int main () {
  try {
    Base* a = new Base;
    Base* b = new Derived;
    cout << "a is: " << typeid(a).name() << '\n';
    cout << "b is: " << typeid(b).name() << '\n';
    cout << "*a is: " << typeid(*a).name() << '\n';
    cout << "*b is: " << typeid(*b).name() << '\n';
  } catch (exception& e) { cout << "Exception: " <<
e.what() << '\n'; }
  return 0;
}
```

```
a is: class Base *
b is: class Base *
*a is: class Base
*b is: class Derived
```

Exceptions

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 20;
  }
  catch (int e)
  {
    cout << "An exception occurred.
Exception Nr. " << e << '\n';
  }
  return 0;
}
```

```
An exception
occurred. Exception
Nr. 20
```

A `throw` expression accepts one parameter (in this case the integer value `20`), which is passed as an argument to the exception 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.

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:

```
try {
  // code here
}
catch (int param) { cout << "int exception"; }
catch (char param) { cout << "char exception"; }
catch (...) { cout << "default exception"; }
```

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. This is done with the expression `throw;` with no arguments. For example:

```
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 instead of looking for a handler or calling std::terminate.

If this throw specifier is left empty with no type, this means that std::unexpected is called for any exception. Functions with no throw specifier (regular functions) never call std::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;
}
```

```
My exception happened.
```

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

exception	description
bad_alloc	thrown by `new` on allocation failure
bad_cast	thrown by `dynamic_cast` when it fails in a dynamic cast
bad_exception	thrown by certain dynamic exception specifiers
bad_typeid	thrown by `typeid`
bad_function_call	thrown by empty function objects
bad_weak_ptr	thrown by shared_ptr when passed a bad weak_ptr

Also deriving from `exception`, header <exception> defines two generic exception types that can be inherited by custom exceptions to report errors:

exception	description
logic_error	error related to the internal logic of the program
runtime_error	error detected during runtime

Preprocessor directives

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 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];
```

After the preprocessor has replaced `TABLE_SIZE`, the code becomes equivalent to:

```
int table1[100];
int table2[100];
```

`#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;
}
```

```
5
7
```

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];
```

This would generate the same code as:

```
int table1[100];
int table2[200];
```

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);
```

This would be translated into:

```
cout << "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 becomes less readable, since the syntax expected is on many occasions different from the normal expressions programmers expect in C++.

Conditional inclusions (#ifdef, #endif, #ifndef, #if, #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
```

In this case, the line of code `int table[TABLE_SIZE];` is only compiled if `TABLE_SIZE` was previously defined with `#define`, independently of its value. If it was not defined, that line will not be included in the program compilation.

`#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];
```

Chained directives

The `#if`, `#else` and `#elif` (i.e., "else if") directives serve to specify some condition to be met in order for the portion of code they surround to be compiled. The condition that follows `#if` or `#elif` can only evaluate constant expressions, including macro expressions. For example:

```
#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];
```

The behavior of `#ifdef` and `#ifndef` can also be achieved by using the special operators `defined` and `!defined` respectively in any `#if` or `#elif` directive:

```
#if defined ARRAY_SIZE
#define TABLE_SIZE ARRAY_SIZE
#elif !defined BUFFER_SIZE
#define TABLE_SIZE 128
#else
#define TABLE_SIZE BUFFER_SIZE
#endif
```

Line control (#line)

Skipped

Error directive (#error)

This directive aborts the compilation process when it is found, generating a compilation error that can be specified as its parameter:

```
#ifndef __cplusplus
#error A C++ compiler is required!
#endif
```

This example aborts the compilation process if the macro name `__cplusplus` is not defined (this macro name is defined by default in all C++ compilers).

Source file inclusion (#include)

When the preprocessor finds an `#include` directive it replaces it by the entire content of the specified header or file.

Input/Output with files

- `ofstream`: Stream class to write on files
- `ifstream`: Stream class to read from files
- `fstream`: Stream class to both read and write from/to files.

```
// basic file operations
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  ofstream myfile;
  myfile.open ("example.txt");
  myfile << "Writing this to a file.\n";
  myfile.close();
  return 0;
}
```

```
[file
example.txt]
Writing this to a
file.
```

Open a file

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

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);
```

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

This member function takes flushes the associated buffers and closes the file:

```
myfile.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;
}
```

```
[file example.txt]
This is a line.
This is another
line.
```

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

```cpp
// 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 is a line.
This is another line.
```