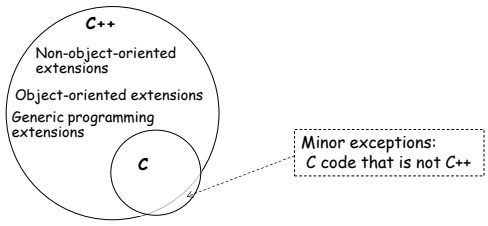
Non-object-oriented features of C++

C++ was developed from the C programming language by adding some features. These features can be collected in three groups:

1. Non-object-oriented features, which can be used in the coding phase. These are not involved with the programming technique (OOP).
2. Features that support object-oriented programming.
3. Features that support generic programming.

With minor exceptions, C++ is a superset of C.



Declarations and Definitions in C++:

There is a difference between a *declaration* and a *definition*.

Declaration:

- A declaration introduces a name – an identifier – to the compiler.
- It provides only the basic attributes of a symbol: its **unique name** and **type**.
- It tells the compiler, "This function or this variable exists somewhere, and here is what it should look like."
- A declaration does not allocate memory space for the name.

Example:

The signature of a function without its body is a declaration.

```
int function (unsigned int, double);   // Declaration (signature)
```

The declaration does not provide the body of the function.

The compiler can still compile a source file (compilation unit) that includes a call to this function.

```
j = function(12, 3.14);   // Declaration is sufficient to compile
```

However, to create an executable code, the body of the function must exist (must be **defined**) somewhere in the program (same or another file).

If the definition of the function is not provided, the linker will generate an error.

Declarations and Definitions in C++ (contd):

Definition:

- A definition is also a declaration. It introduces the name and type.
- In addition, a definition provides all of the necessary information to create that entity (variable, function, class) in its entirety.
- For example:

  Defining a function means providing a function body;

  Defining a class means giving all of the variables and methods of the class.

All definitions are declarations, but not all declarations are definitions.

Declaring an identifier (variable, function, class) without defining it is necessary and useful, especially if you work **with multiple source files** and you need to use the same name (for example, a function) across them.

There is no need to put the body of a function in multiple files, but it does need to be declared in each file where it is used.

The definition of an identifier (for example, the body of the function) will take place only in one file (one definition rule).

Often, the *compiler* only needs a declaration for something to compile a file into an object file, expecting that the *linker* can find the definition from another file.

Declarations and Definitions in C++ (contd):

Examples:

```
extern int i;          // Declaration, the definition is in another file
int i;                 // Definition, memory is allocated
-----------------------
struct ComplexT;       // Declaration only
struct ComplexT{  // Declaration (type) and definition of complex numbers
  double re{}, im{};
};
ComplexT c1,c2;     // Definition of two complex number variables c1, c2
-----------------------
void function(int, int);   // Declaration (its body is the definition)
void function(int, int){   // Definition
 :
}
-----------------------
class Point;             // Declaration only
class Point{             // Declaration and Definiton of Point Class
 public:
   void move(int, int);  // Declaration of the function to move the Points
 private:
   int x{}, y{};         // Definition of the properties: x and y coordinates
};
Point point1, point2;    // Definition of two Point objects
```


The One Definition Rule (ODR):

In a compilation (translation) unit (source file), **no** variable, function, class type, enumeration type, or template must ever be ***defined* more than once**.

You can have more than one *declaration* for any entity, but there must always be **only *one definition*** that determines what it is and causes it to be created.

If there is more than one definition within the same translation unit, the code will not compile.

The ODR rule also applies to *an entire program*.

No two definitions of the same identifier are allowed, even if they're identical and appear defined in different translation units.

When we work with multiple files, we must declare an identifier in each file where it is used because the compiler needs to have a declaration of identifiers to compile a source file into an object file.

- **An identifier can be declared as often as you want.**
- **However, it must be defined exactly once in a program.**

If you define something more than once (even in different files), the linker generates a linker error (*duplicate symbols*).

If you forget to define something that has been declared and referenced, the linker also generates a linker error (*missing symbol*).

Namespaces

When a program reaches a certain size, it is usually divided into pieces, each built and maintained by a separate developer or group (teamworking).

Since **C++ uses a single arena** for all identifiers and functions, developers must avoid accidentally using the same name in conflicting situations.

A programmer faces the same problem if she uses the same names as library functions.

Standard C++ has a mechanism to prevent this collision: the **namespace** keyword.

Each set of C++ definitions in a library or program is "wrapped" in a namespace.

If another definition has an identical name but is in a different namespace, there is no collision.

Example:

```
namespace programmer1{          // programmer1's namespace
     int iflag;                 // programmer1's iflag

     void g(int);               // programmer1's g function
     :                          // other variables
}                               // end of namespace

namespace programmer2{          // programmer2's namespace
     int iflag;                 // programmer2's iflag
     :
}                               // end of namespace
```

Accessing the variables defined in namespaces:

The scope operator "::" is used to access the variables defined in namespaces.

```
programmer1::iflag = 3;       // programmer1's iflag
programmer2::iflag = -345;    // programmer2's iflag
programmer1::g(6);            // programmer1's g function
```

If a variable or function does not belong to any namespace, it is defined in the global namespace.

It can be accessed without a namespace name and scope operator.

Comments:

- A well-written code should explain itself.
- Simply repeating the code in a comment is considered bad practice.

For example, `programmer1::iflag = 3;` *//programmer1's iflag is 3* ***(useless)***

- Since the objective of this lecture slides is teaching programming, code comments are used to explain even the most basic lines of C++ code.
- You should only include comments in your code that clarify or document aspects that may not be immediately clear to the reader, such as yourself or your coworkers.

using declaration:

This declaration makes accessing variables and functions defined in a namespace easier.

```
using programmer1::iflag;   // applies to a single item in the namespace
```

After this declaration, to access the variable `iflag` in the namespace `programmer1`, writing the name of the namespace is not necessary.

```
iflag = 3;                  // programmer1::iflag=3,
programmer2::iflag = -345;  // namespace name is necessary
programmer1::g(6);          // namespace name is necessary
```

This declaration can also apply to all elements in a namespace.

```
using namespace programmer1;  // applies to all elements in the namespace
iflag = 3;                    // programmer1::iflag = 3;
g(6);                         // programmer1::g(6);  programmer1's function g
programmer2::iflag = -345;
```


Working with multiple files (Separate compilation)

As our code base grows, creating separate files for related entities (constants, variables, functions, classes) is a proper approach.

- It provides managing the complexity of the software and reusability of entities in new projects.
- We need to compile only the necessary files whenever the code is changed.

Prior to C++20, programs were organized in ***header files*** and source files.

This approach has some disadvantages, such as creating issues with ODR and increasing the size of the source code.

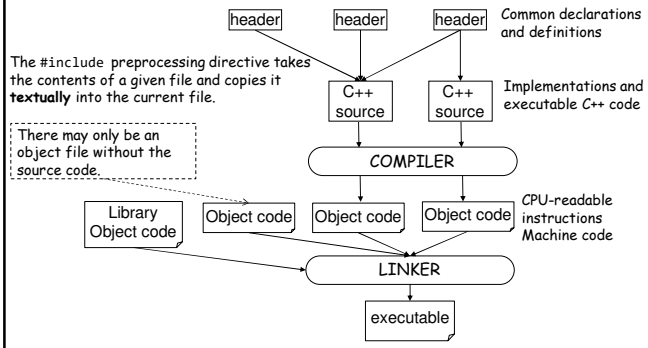
C++20 introduced ***modules*** that solve problems generated by header files and reduce build (compilation) times, especially in large codebases with many dependencies.

Since header files are still widely used, first we will discuss them briefly.

Then we will cover C++ modules.

Working with multiple files using header files:

The header files contain the common declarations and definitions.



Standard C++ header files:

In the first versions of C++, mainly '**.h**' is used as the extension for the header files of the standard library.

As C++ evolved, different compiler vendors chose other extensions for file names (`.hpp`, `.H`, etc.). These issues caused source code portability problems.

To solve these problems, the standard uses a format that allows file names longer than eight characters and eliminates the extension for the header files of the standard library.

For example, instead of the old style of including **iostream.h**, which looks like this: `#include <iostream.h>` **X**

you can now write: **`#include <iostream>`** √

The libraries inherited from C are still available with the traditional '**.h**' extension. However, you can also use them with the more modern C++ include style by putting a "**c**" before the name. Thus:

```
#include <stdio.h>      become:     #include <cstdio>
#include <stdlib.h>                 #include <cstdlib>
```


Standard C++ header files (contd):

Today most C++ compilers support old libraries and header files too. So you can also use the old header files with the extension '`.h`'.

For a high-quality program, always prefer the new libraries and use standard header files without extension.

You may still use the extension '`.h`' for your own header files.

Example: `#include "myheader.h"`

Disadvantages of using header files:

- They increase the size of the source code and slow compilation because when multiple files include the same header file, it is reprocessed multiple times.
- The order of `#includes` can modify behavior or break code.
- They may cause issues with the ODR because the same definition may (and must) be included multiple times. Any definition you place in a header gets copy-pasted into every translation unit that includes it, either directly or indirectly.

The C++20 standard introduces ***modules*** as a novel way of structuring C++ libraries and programs as components.

Many of the problems associated with header files are eliminated or reduced by using modules.

Working with Multiple Files (Separate Compilation) (contd)

Working with multiple files (*separate compilation*) requires a method to compile each file automatically.

Further, it is necessary to instruct the linker to build all the pieces, along with the appropriate libraries and startup code, into an executable.

The solution, developed on Unix but available everywhere in some form, is a program called **make**.

Compiler vendors have also created their own project-building tools.
Generally, these tools use a project file similar to a `makefile`, but the programming environment maintains this file.

The configuration and use of project files vary from one development environment to another, so you must find the appropriate documentation for using them.

We will examine two examples, i.e., e02_1a.zip and e02_1b.zip, which illustrate how to work with multiple files a few slides later.

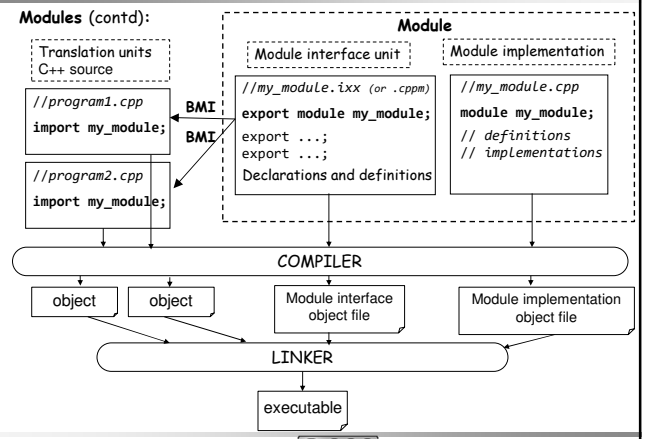
The example e02_1a.zip uses header files.

The example e02_1b.zip uses modules.

Modules (Since C++20):

- The use of modules eliminates the need for header files.
  Modules are not inserted textually into source files.
- Module interfaces are precompiled and cached (stored in memory) for shorter compilation times.
- The content of a module interface file is never duplicated, even if multiple source files use it.
  The precompiled result of an importable module unit (Built Module Interface - **BMI**) can be consumed (used) by multiple source files.
- Using modules decreases the possibility of running into ODR (One Definition Rule) issues.
- Instead of re-compiling and linking the entire codebase every time a small change is made, only the changed module and modules that depend on it must be recompiled.
  This can significantly reduce build times, especially in large codebases with many dependencies.
- Additionally, C++ modules can help reduce the amount of code that needs to be recompiled by resolving dependencies at compile time and enabling more fine-grained control over what parts of a codebase need to be rebuilt.

Modules (contd):



Creating Modules:

We typically create a module for the set of code that encompasses a specific purpose.

Each module would represent a logical grouping of types, functions, and relevant global variables.

A module can ***export*** any number of C++ entities (constants, functions, types, and so on), which can then be used in any source file that ***imports*** that module.

A module consists of two files, i.e., module ***interface*** and module ***implementation***.

The module interface file:

This file contains declarations (signatures) of functions and definitions of types (classes) and, if necessary global data (usually constants).

Example:

Filename. The Visual Studio suggests the extension ixx.
Some compiler vendors use .cppm as extension.
The filename can be different than the module name.

```
// functions.ixx
export module functions;  // The name of the module is functions

export const double PI{ 3.14 };    // Definition of a constant double
export double function1(double);   // Declaration of a function
export int function2(int);         // Declaration
```


Creating Modules (contd):

The module implementation file:

This file contains definitions (implementations) of functions.

Example (contd):

```
// functions.cpp       Filename can be different from the module name
module functions;     // Module name is functions
// function1 increments the input parameter by 0.1
double function1(double input) {
  return input + 0.1;
}
// function2 increments the input parameter by 1
int function2(int input) {
  return input + 1;
}
```

The bodies of the functions could also be provided in the module interface file.

In this case, the implementation file would not be necessary.

However, separating the interface and implementation is a good practice.

Using Modules (contd):

The consumer (user) source files can import the modules and use the exported entities.

Example (contd):

Module name, not the file name

```
// main.cpp
import functions;            // Importing the module: functions
int main()
{
  double d { PI };       // A double number is defined and initialized to PI
  d = function1(d);      // function1 is imported from the module functions
  int i{};
  i = function2(i);      // function2 is imported from the module functions
  return 0;
}
```

Header Units:

A header unit is a binary representation of a header file that can be imported as a module.

Example:

```
import <iostream>;   // for IO operations
```

Header units are a step in between header files and C++ 20 modules.

The C++ standard library modules (Since C++ 23):

The C++23 standard library introduces a module: **std**, that exports the declarations and names defined in the C++ standard library namespace `std`, such as `std::cout`, `std::print()`, and `std::string`.

It also exports the contents of C wrapper headers such as `<cstdio>` and `<cstdlib>`, which provide functions like `std::printf()`.

Example:

```
import std;        // module of standard library; Since C++23
int main() {
  std::string str { "ABC" };
  std::cout << str;
   :
```


The namespace `std`

In the standard library of C++, all declarations and definitions take place in the namespace: **`std`**

You should be aware of namespaces if you use standard headers.

For example, `std::string`, `std::cout`, `std::vector`, `std::sort` etc.

Probably, you will write `using namespace std;` at the beginning of your file to avoid writing `std::` repeatedly.

However, the statement `using namespace std;` is generally considered bad practice.

It increases the risk of name conflicts. `std` is an extensive library, and you may use the same names already defined in the library.

Moreover, it is helpful to know which identifiers (variables, functions) are defined by the developer and which are taken from the library.

Suggestions:

- Import only some well-known identifiers: **`using std::cout; using std::cin;`**
- If you still import entire namespaces, do so inside classes, functions, or limited scope and not in the global scope.
- You may import your own namespaces entirely: `using namespace my_namespace;`

Input / Output

When a C++ program includes (or imports since C++20) the **iostream** header or imports **std** module (introduced in C++23), four objects are created and initialized:

- **cin** handles input from the standard input, the keyboard.
- **cout** handles output to the standard output, the screen.
- **cerr** handles unbuffered output to the standard error device, the screen.
- **clog** handles buffered error messages to the standard error device

Using cout object: To print a value to the screen, we use the predefined object **cout**, and the insertion operator (<<).

Using cin object: The predefined **cin** stream object is used to read data from the standard input device, usually the keyboard. The **cin** stream uses the >> operator, usually called the "get from" operator.

```
import std;  // or #include<iostream> prior to C++20 or import <iostream>;
int main() {
 int i, j;                              // Two integers are defined
 std::cout << "Give two integers \n";   // Message to screen, to the new line
 std::cin >> i >> j;                    // Read i and j from the keyboard
 std::cout << "Sum= " << i + j << "\n"; // The sum to the screen
 return 0;
}
```

Headers: Example e02_1a.zip

Input / Output (contd)

Starting from C++23, the preferred way to **output text** to the computer's screen is by using functions like **`std::println()`** and **`std::print()`**.

Example:

```
import std;
int main() {
  int i, j;                              // Two integers are defined
  std::println("Give two integers");     // Message to screen, goto new line
  std::cin >> i >> j;                    // Read i and j from the keyboard
  std::println("Sum=  {}", i + j );      // The sum to the screen
  return 0;
}
```

Modules: Example e02_1b.zip

Replacement field

The only difference with `std::println()` is that `std::print()` does not add a "new line" break (`\n`) at the end.

You cannot invoke `std::print()` or `std::println()` without a format string.

For example, you cannot use `std::println(i+j)` to output only the value of the sum.

Instead, you use a statement of the form `std::println("{}", i + j)`.

Initializing variables

There are three mechanisms for initializing a variable: ***functional notation***, ***assignment notation***, and ***uniform initialization (curly braces)***.

```
unsigned int number_of_students(100);       // Functional notation
unsigned int number_of_courses = 12;        // Assignment notation

// Uniform initialization (curly braces)
unsigned int car_count {10};                // Number of cars
unsigned int bus_count {5};                 // Number of busses
unsigned int total_vehicle {car_count + bus_count}; //Total (vehicles)
```

The braced initializer form is safer if there is a *narrowing conversion*.
A narrowing conversion changes a value to a type with a more limited range of values.

```
unsigned int car_count(10.3);  // car_count = 10  (There is a warning)
unsigned int bus_count = 5.6;  // bus_count = 5   (There is a warning)
unsigned int car_count {10.3}; // Compile Error!
```

The main advantage of braced initialization is that it allows programmers to initialize just about everything in the same manner.

Thus, it is also known as ***uniform initialization***.

Later, we will also use it to initialize objects.

Initializing variables (contd)

Zero Initialization:

The following statement defines an integer variable with an initial value equal to zero:

```
int counter {0};      // counter starts at zero
```

You could omit the 0 in the braced initializer:

```
int counter {};       // counter starts at zero
```

Suggestion:

You don't have to initialize variables when you define them.

However, it is a good idea to ensure that variables start with known values.

This makes it easier to determine what is wrong when the code doesn't work as you expect.

Type deduction using the `auto` keyword

In C++, we can use the `auto` keyword to let **the compiler deduce** the type of a variable from the **initial values** we supply.

Examples:

```
auto v1 {10};      // The type of v1 is int
auto v2 {2000UL}; // The type of v2 is unsigned long (at least 32 bits)
auto v3 {3.5};     // The type of v3 is double
```

We can also use functional or assignment notation with `auto` for the initial value.

Examples:

```
auto v1 = 10;      // The type of v1 is int
auto v2 = 2000UL; // The type of v2 is unsigned long (at least 32 bits)
auto v3(3.5);      // The type of v3 is double
```

The type deduction occurs exclusively at **compile time**.

The type must be clear to the compiler based on the provided initial value.

In C++, the type of a variable **cannot** be deduced at runtime.

Type deduction using the `auto` keyword (contd)

- It is recommended to explicitly specify the type (do not use `auto`) when defining variables of fundamental types like `char`, `int`, `double`, etc.
  This increases the understandability of your program.
- When type names are complicated (verbose or long), you can use the `auto` keyword to increase the readability of your code.
- You can use the `auto` keyword as the return type of a function when you do not want to specify the return type explicitly.
  Example:
  `auto function1(int, double);`
  By considering the `return` statements in the definition of the function, the compiler will deduce the return type of this function.

The keyword `auto` never deduces to a reference type, always to a value type.

To have the compiler deduce a reference type, you should write **`auto&`** or **`const auto&`**.

We will cover the details in the coming chapters.

The Lifetime and the Scope of a Variable:

All variables have a finite ***lifetime***.

They are created at the point at which they are defined, and at some point, they are destroyed, at the latest, when the program is terminated.

There are four different kinds of storage duration (**lifetime**):

1. **Automatic storage duration:** Standard variables are defined within a block without using the static keyword.
   They exist from the point at which they are defined until the end of the block, which is the closing curly brace "}".
   Automatic variables have ***local scope*** or ***block scope***.
2. **Static storage duration:** Variables are defined using the **`static`** keyword. Static variables exist from the point at which they are defined and continue in existence until the program ends.
3. **Dynamic storage duration:** For these variables, memory is allocated at runtime.
   They exist from the point at which you create them until you release their memory to destroy them (remember: `new`, `delete`, pointers).
4. **Thread storage duration:** Variables are declared with the **`thread_local`** keyword (*for parallel programming*).
   Thread local variables are out of the scope of this course.

Scope of a variable:

The **scope** of a variable is the region of a program in which the variable name is valid.

- **Within its scope**, you can set or read the variable's value.
- **Outside of its scope**, you cannot refer to its name. Any attempt to do so will result in a compiler error.

Note that a variable may still exist outside of its scope, even though you cannot refer to it.

We will see examples of this situation later when we cover variables with static and dynamic storage duration.

Summary:

Lifetime: The period of execution time over which a variable exists.

Scope: The region of program code over which the variable name can be used.

Global Variables:

Variables defined outside of all blocks and classes are also called **globals** and have **global scope** (also called global namespace scope).

Global variables are accessible in all the functions in the source file following the point at which they're defined.

Global variables have static storage duration by default, so they exist from the start of the program until the execution of the program ends.

Avoid global variables! Common coding and design guidelines suggest that global variables should be avoided.
Declaring all variables in global scope increases the possibility of accidental, erroneous modification of a variable.

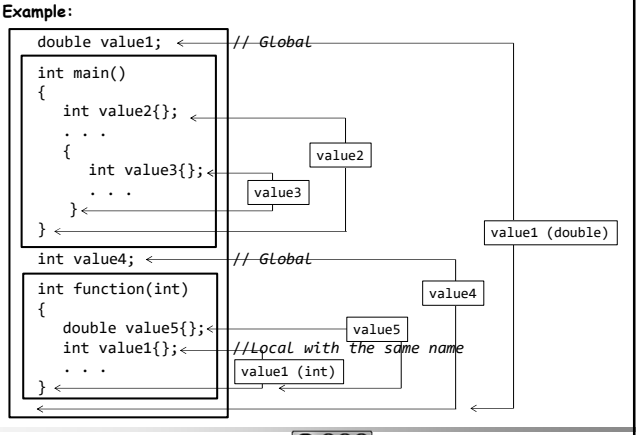
As a result, it is difficult to determine which part of the code is responsible for changing global variables.

Moreover, global variables occupy memory for the duration of program execution, so the program will require more memory than if you used local variables.

Global variables declared with the **`const`** keyword are an exception to this rule.

It is recommended to define all your constants only once, and global variables are ideally suited for that.

Example:



Scope Resolution Operator (::)

A definition in a block (local name) can hide a definition in an enclosing block or a global name.

It is possible to access a hidden global variable by using the **scope resolution operator** (::)

```
int y = 0;            // Global y
int x = 1;            // Global x
void f(){             // Function is a new block
   int x=5;           // Local  x=5, it hides global x
   ::x++;             // Global x=2
   x++;               // Local x=6
   y++;               // Global y=1, scope operator is not necessary
}
```

Example e02_2.cpp

It is not recommended to give identical names to global and local data if it is not mandatory.

Like in C, in C++, the same operator may have more than one meaning.

The scope operator :: also has many different tasks, which are presented in the following chapters.

Constants

C++ introduces the concept of a named constant that is just like a variable, except that its value cannot be changed.

The modifier **const** tells the compiler that a name represents a constant.

```
const int MAX = 100;  // MAX is constant, and its value is 100
```
or
```
const int MAX(100);   // MAX is constant, and its value is 100
```
or
```
const int MAX {100};  // MAX is constant, and its value is 100
```

The following statement causes a compiler error if MAX is a constant.

```
MAX = 5;              // Compiler Error!  Because MAX is constant
```

const can take place before (left) and after (right) the type. They are always (both) allowed and equivalent.

```
int const MAX {100};       // The same as const int MAX {100};
```

The keyword const very often occurs in **C++** programs, as we will see in this course. **This usage decreases error possibilities.**

To make your programs more readable, use uppercase font for constant identifiers.

Using the const keyword in the declaration of pointers.

There are three different cases:

1) The data (pointed to by the pointer) **is constant**, but the pointer itself may be changed.

```
const char *ptr = "ABC";  //Constant data = "ABC", pointer is not const
```
or
```
const char *ptr {"ABC"};  //Constant data = "ABC", pointer is not const
```

Here, ptr is a pointer variable, which points to chars.

The const word may also be written after the type:

```
char const * ptr {"ABC"}; // Constant data = "ABC", pointer is not const
```

Whatever is pointed to by ptr may not be changed because the chars are declared as const.

The pointer ptr itself, however, may be changed.

```
*ptr = 'Z';    // Compiler Error! Because data is constant
ptr++;         // OK, because the address in the pointer may change.
```


2) The pointer itself is a const pointer which may not be changed. Data pointed to by the pointer may be changed.

```
int data {10};                    Address of data
int * const cp {&data};   // Pointer is constant, data may change
*cp = 15;                 //OK, data is not constant
cp++;                  //Compiler Error! Because the pointer is constant
```

3) Neither the pointer nor what it points to may be changed

```
const double data {1.2};
const double * const ccp {&data};    // Pointer and data are constant
*ccp = '2.3';              //Compiler Error! Because data is constant
ccp++;                     // Compiler Error! Because pointer is const
```

The same pointer definition may also be written as follows:

```
double const * const ccp {&data};
```

The definition or declaration in which const is used should be read from the variable or function identifier back to the type identifier:

"ccp is a const pointer to const double data" .

inline Functions

In C, macros are defined by using the #define directive of the preprocessor.

In C++, instead of function-like macros, inline functions are used. Here, the keyword **inline** is inserted before the declaration of a standard function.

The difference between standard functions and inline functions (macros in C):

A **standard function** is placed in a separate section of code, and a call to the function generates a jump to this section of code.

- The advantage of this approach is that the same code can be called (executed) from many different places in the program. This makes it unnecessary to duplicate the function's code every time it is executed.
- However, there is also a disadvantage.
  - The function call itself, and the transfer of the arguments takes some time.
  - Before the jump, the return address and arguments are saved in memory (usually in the stack).
  - When the function has finished executing, the return address and return value are taken from memory, and the control jumps back to the statement following the function call.
  - In a program with many function calls (especially inside loops), these times can add up and decrease the performance.

inline Functions (contd)

An **inline function** is defined using almost the same syntax as an ordinary function.

However, instead of placing the function's code in a separate location, the compiler simply inserts the **machine-language code** into the location of the function call.

Using inline functions increases the size of the executable code.
However, the program may run faster because transferring parameters and the return address is unnecessary.

```
inline int max (int i1, int i2){  // An inline function
     return(i1 > i2) ?  i1 : i2;  // returns the greatest of two integers
}
```

Calls to the function are made in the normal way:

```
int j, k, l ;          // Three integers are defined
……..                  // Some operations over k and l
j = max( k, l )        // inline function max will be inserted here
```

The decision to inline a function must be made with some care.

It's appropriate to inline a function when it is short, but not otherwise.

If a long or complex function is inlined, too much memory will be used, and not much time will be saved.

Default Function Arguments

A programmer can give default values to the parameters of a function. In calling the function, default values are used if the arguments are not provided.

Example:

```
void f(char c, int i1=1, int i2=2)  // i1 and i2 have default values
{ … }                               // Body of the function
```

This function may be called in different ways:

```
f('A',4,6);     // c='A', i1=4, i2=6
f('B',3);       // c='B', i1=3, i2=2
f('C');         // c='C', i1=1, i2=2
f('C', {}, 7);  // c='C', i1=0, i2=7
```

In calling a function argument must be given from left to right:

```
f('C', ,7); // ERROR! The third argument is given, but the second is not
```

While defining functions, default values of parameters must be given from right to left without skipping any parameter.

```
void f(char c='A', int i1, int i2=1)   // ERROR! i1 has been skipped
```

Default values must not be only constant values. They may also be expressions or function calls.

```
void f(char c, int i1 = other_func())
```


Overloading of function Names

C++ enables several functions of the *same name* to be defined as long as these functions have different sets of parameters (Numbers, types, or the order of the parameters may be different).

The name and the parameter list build the *signature* of the function.

Example:

```
// Structure for complex numbers
struct ComplexT{
  float re, im;
};
// print function for real numbers
void print (float value){
  std::println("value= {}", value);
}
// print function for complex numbers
void print (ComplexT c){
  std::println("real= {}  im= {}", c.re, c.im);
}
// print function for real numbers and characters
void print (float value, char c){
   std::println("value= {} c= {}", value, c);
}
```

```
int main()
{
   ComplexT z;
   z.re=0.5;
   z.im=1.2;
   print(z);
   print(4.2);
   print(2.5,'A');
   return 0;
}
```

See Example e02_3.cpp

See Example e02_3_with_cout.cpp

Reference Operator (&)

This operator provides an alternative name for storage.

```
int i = 5;     // integer i = 5
int& j = i;  // j is a reference to i. j and i both have the same address
j++;         // i = 6
```

Actually, there is only **one integer memory cell** with two names: i and j.

It is often used to pass parameters to a function by their references.

Remember: We can also use curly braces to initialize variables.

```
int i {5};  // integer i = 5
int& j {i}; // j is a reference to i. j and i both have the same address
j++;        // i = 6
```


Call by reference:

Remember: we can pass parameters to functions either by their *values* or by their *addresses* using pointers.

There are two main reasons for passing parameters by their addresses:

1. To modify the original value of a parameter in the function.
2. To avoid extensive data being copied into the stack (memory).

Case 1:

If we want that the function can modify the original value of a parameter, then we must send its address to the function.

Example: (Call by value) The function cannot modify the original value of the parameter

```
void calculate(int j) {
   j = j * j / 2;      // j cannot be changed,  function is useless
}
int main()
  {
      int i = 5;
      calculate(i);   // i cannot be modified
      return 0;
  }
```


Case1 (contd):

Solution with pointers (C Style):

```
void calculate(int *j) {
    *j = *j * *j/2;              // Difficult to read and understand
}                                // * has multiple meanings
int main()
{
   int i{5};
   calculate(&i);              // Address of i is sent
   return 0;
}
```

Call by address

Here the symbol & is not the **reference operator**; it is the **address operator**.

C-style solution using pointers is difficult to read and understand.

Case1 (contd):

Solution with references (C++ Style):

```
void calculate(int& j) {    // j is a reference to the coming argument,
                            // two variables have the same address
    j = j*j/2;              // In the body, j is used as a normal variable
}

int main( )
{
     int i{5};
// A normal function call.
// However, instead of the value, the address of i is sent
     calculate(i);
     return 0;
}
```

Call by reference

The solution using the reference operator is easier to read and understand.

Case2:

Another reason for passing parameters by their address is to avoid extensive data being copied into the stack (memory).

Remember that all arguments sent to a function are copied into the stack. This operation takes time and wastes memory.

If we need to send large data to a function, we prefer to send its address using the reference operator instead of its value.

To prevent the function from changing the parameter accidentally, we pass the argument as a **constant reference** to the function.

Example:

We store data about persons that consists of two parts, i.e., name (40 characters) and reg_num (unsigned int: 4 bytes). The total is 44 bytes.

```
struct Person{                  // A structure to define persons
    char name [40];             // Name 40 bytes (use std::string type)
    unsigned int reg_num;       // Register number 4 bytes
};                              // Total: 44 bytes
```

The size of the integers and addresses may depend on the system where the program runs.

Case2 (contd):

```
struct Person{                  // A structure to define persons
    char name [40];             // Name 40 bytes (use std::string)
    unsigned int reg_num;       // Register number 4 bytes
};

void print (const Person& per) // per is constant reference parameter
{
    std::println("Name: {}", per.name);     // name to the screen
    std::println("Num: {}", per.reg_num);   // reg_num to the screen
}

int main(){
    Person ahmet;                       // ahmet is a variable of type Person
    strcpy(ahmet.name,"Ahmet Bilir");       // name = "Ahmet Bilir"
    ahmet.reg_num = 324;                    // reg_num= 324
    print(ahmet);                           // Function call
    return 0;
}
```

Instead of 44 bytes, only 4 bytes (address) are sent to the function.

The size of the integers and addresses may depend on the system where the program runs.

In any case, if you use **large data**, call-by-reference will transfer fewer bytes than call-by-value.

Return by reference:

By default in C++, when a function returns a value: return *expression*;

Expression is evaluated, and its value is copied into the stack. The calling function reads this value from the stack and copies it into its variables.

An alternative to "return by value" is "return by reference", in which the value returned is not copied into the stack. The address is returned.

One result of using "return by reference" is that the function which returns a parameter by reference can be used on the left side of an assignment statement.

The calling function can modify the returned value.

Example: This function returns a reference to the largest element of an array.

```
int&  max(int a[], unsigned int length) // Returns a reference to int
{
    …                           // Find the largest element of a[]
    return a[i];                // Returns reference to a[i]
}
int main()
{
   int array[ ] = {12, -54 , 1 , 123, 63}; // An exemplary array
   max(array,5) = 0;                // write 0 over the largest element
   :
```

See Example e02_4.cpp

Return by reference (contd):

- To prevent the max function from changing the input array accidentally, and
- To prevent the calling function from changing the return parameter accidentally, we can use **const** qualifiers.

If a function returns a constant reference, it cannot be used on the left side of an assignment.

Output is constant — Input is constant

```
// max function cannot be used on the left side of an assignment statement
const int&  max(const int a[], unsigned int length)
{
   ...                      // Find the largest element of a[]
   return a[i];             // Returns reference to a[i]
}
```

This function can only be on the right side of an assignment.

```
int main()
{
   ...
   largest = max(array,5);   // Get the largest element
   max(array,5) = 0;         // Compiler ERROR! Constant reference
}
```

See Example e02_5.cpp

Never return an automatic local variable by reference!

Remember: When a function returns, local variables go out of existence, and their values are lost.

Since a function that uses "return by reference" returns an actual memory address, the variable in this memory location must remain in existence after the function returns.

```
int&  f( )                  // Return by reference
{
  int i;                    // Local variable. Created in stack
   :
  return  i;                // Caution! i does not exist anymore.
}
```

In this case, the compiler may only output a warning message, and you may run this code.

Furthermore, sometimes you can get correct results if a new variable does not use the related memory location.

However, your program will not be reliable.

Operator Overloading

In C++, it is also possible to overload the built-in C++ operators such as +, -, =, and ++ so that they invoke different functions depending on their operands.

That is, the + in a+b will add the variables if a and b are integers but will call a different function if a and b are variables of a user-defined type.

Some rules:

- You cannot overload operators that do not already exist in C++.
- You can not change the number of operands. A binary operator (for example, +) must always take two operands.
- You can not change the precedence of the operators.

  For example, * comes always before +

Everything you can do with an overloaded operator, you can also do with a function.

However, by making your listing more intuitive, overloaded operators make your programs easier to write, read, and maintain.

Operator overloading is mainly used with objects. We will discuss this topic later in more detail.

Writing functions for operators:

Functions of operators have the name `operator` and the symbol of the operator. For example, the function for the operator + will have the name `operator+` .

Example: Overloading of operator (+) to add complex numbers:

```
struct ComplexT{       // Structure for complex numbers
     float re,im;
};

// Function for overloading of operator (+) to add complex numbers
ComplexT operator+ (const ComplexT& v1, const ComplexT& v2){
    ComplexT result;                       // Local result
    result.re = v1.re + v2.re;
    result.im = v1.im + v2.im;
    return result;
}

int main(){
  ComplexT c1, c2, c3;  // Three complex numbers
  c3 = c1 + c2;        // The function is called. c3 = operator+(c1,c2);
  return 0;
}
```

See Example e02_6.cpp