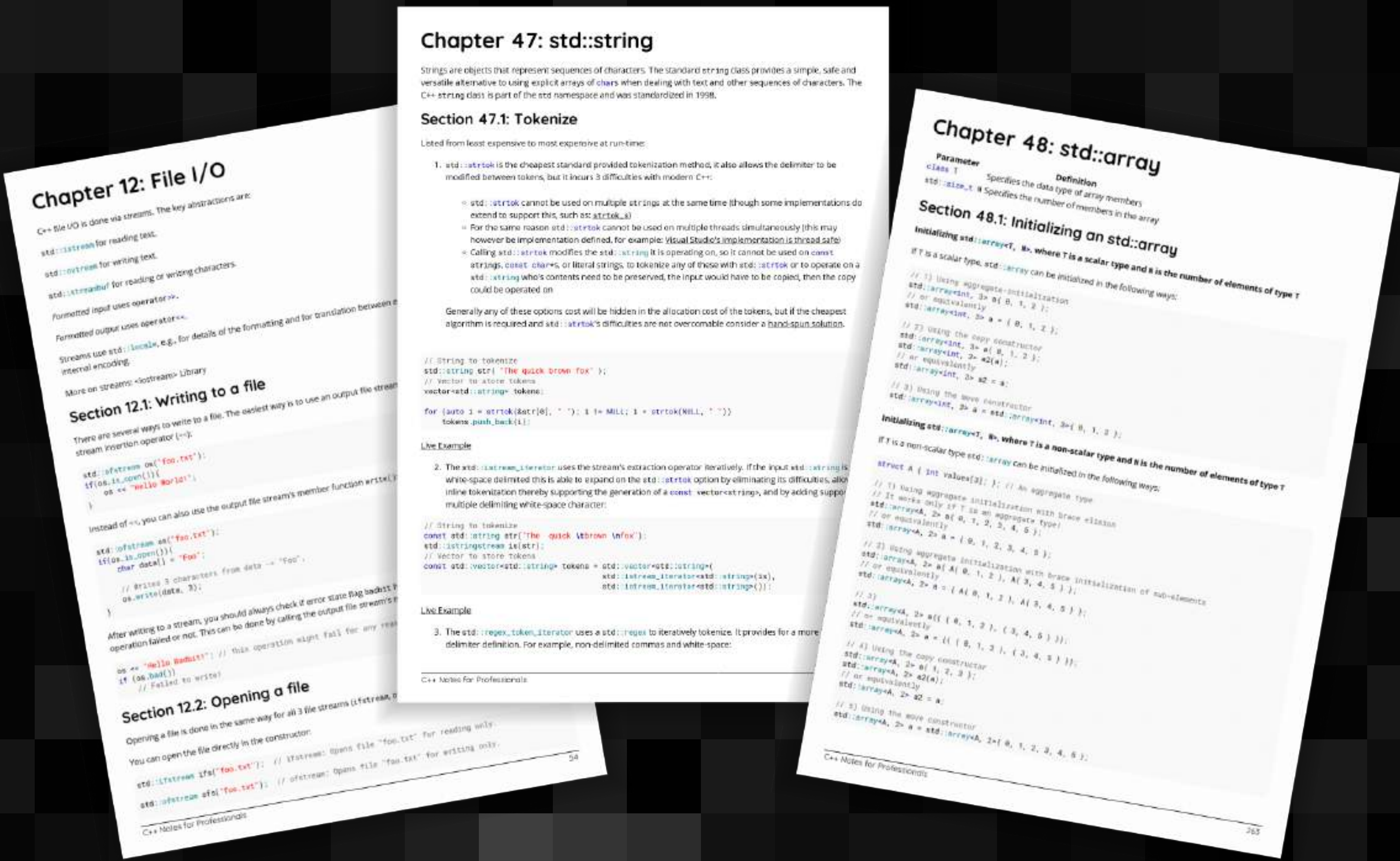
C++

Notes for Professionals



600+ pages

of professional hints and tricks

GoalKicker.com
Free Programming Books

Disclaimer
This is an unofficial free book created for educational purposes and is not affiliated with official C++ group(s) or company(s).
All trademarks and registered trademarks are the property of their respective owners

Contents

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
https://goalkicker.com/CPlusPlusBook

This *C++ Notes for Professionals* book is compiled from Stack Overflow Documentation, the content is written by the beautiful people at Stack Overflow. Text content is released under Creative Commons BY-SA, see credits at the end of this book whom contributed to the various chapters. Images may be copyright of their respective owners unless otherwise specified

This is an unofficial free book created for educational purposes and is not affiliated with official C++ group(s) or company(s) nor Stack Overflow. All trademarks and registered trademarks are the property of their respective company owners

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Please send feedback and corrections to web@petercv.com

Chapter 1: Getting started with C++

Version	Standard	Release Date
C++98	ISO/IEC 14882:1998	1998-09-01
C++03	ISO/IEC 14882:2003	2003-10-16
C++11	ISO/IEC 14882:2011	2011-09-01
C++14	ISO/IEC 14882:2014	2014-12-15
C++17	TBD	2017-01-01
C++20	TBD	2020-01-01

Section 1.1: Hello World

This program prints `Hello World!` to the standard output stream:

```
#include <iostream>

int main()
{
    std::cout << "Hello World!" << std::endl;
}
```

See it live on Coliru.

Analysis

Let's examine each part of this code in detail:

- `#include <iostream>` is a **preprocessor directive** that includes the content of the standard C++ header file `iostream`.

  `iostream` is a **standard library header file** that contains definitions of the standard input and output streams. These definitions are included in the `std` namespace, explained below.

  The **standard input/output (I/O) streams** provide ways for programs to get input from and output to an external system -- usually the terminal.

- `int main() { ... }` defines a new function named `main`. By convention, the `main` function is called upon execution of the program. There must be only one `main` function in a C++ program, and it must always return a number of the `int` type.

  Here, the `int` is what is called the function's return type. The value returned by the `main` function is an **exit code.**

  By convention, a program exit code of `0` or `EXIT_SUCCESS` is interpreted as success by a system that executes the program. Any other return code is associated with an error.

  If no `return` statement is present, the `main` function (and thus, the program itself) returns `0` by default. In this example, we don't need to explicitly write `return 0;`.

  All other functions, except those that return the `void` type, must explicitly return a value according to their return type, or else must not return at all.

- `std::cout << "Hello World!" << std::endl;` prints "Hello World!" to the standard output stream:

  - `std` is a namespace, and `::` is the **scope resolution operator** that allows look-ups for objects by name within a namespace.

    There are many namespaces. Here, we use `::` to show we want to use `cout` from the `std` namespace. For more information refer to Scope Resolution Operator - Microsoft Documentation.

  - `std::cout` is the **standard output stream** object, defined in `iostream`, and it prints to the standard output (`stdout`).

  - `<<` is, *in this context*, the **stream insertion operator**, so called because it *inserts* an object into the *stream* object.

    The standard library defines the `<<` operator to perform data insertion for certain data types into output streams. `stream << content` inserts `content` into the stream and returns the same, but updated stream. This allows stream insertions to be chained: `std::cout << "Foo" << " Bar";` prints "FooBar" to the console.

  - `"Hello World!"` is a **character string literal**, or a "text literal." The stream insertion operator for character string literals is defined in file `iostream`.

  - `std::endl` is a special **I/O stream manipulator** object, also defined in file `iostream`. Inserting a manipulator into a stream changes the state of the stream.

    The stream manipulator `std::endl` does two things: first it inserts the end-of-line character and then it flushes the stream buffer to force the text to show up on the console. This ensures that the data inserted into the stream actually appear on your console. (Stream data is usually stored in a buffer and then "flushed" in batches unless you force a flush immediately.)

    An alternate method that avoids the flush is:

    ```
    std::cout << "Hello World!\n";
    ```

    where `\n` is the **character escape sequence** for the newline character.

  - The semicolon (`;`) notifies the compiler that a statement has ended. All C++ statements and class definitions require an ending/terminating semicolon.

Section 1.2: Comments

A **comment** is a way to put arbitrary text inside source code without having the C++ compiler interpret it with any functional meaning. Comments are used to give insight into the design or method of a program.

There are two types of comments in C++:

Single-Line Comments

The double forward-slash sequence `//` will mark all text until a newline as a comment:

```
int main()
{
```

```
    // This is a single-line comment.
    int a;  // this also is a single-line comment
    int i;  // this is another single-line comment
}
```

C-Style/Block Comments

The sequence `/*` is used to declare the start of the comment block and the sequence `*/` is used to declare the end of comment. All text between the start and end sequences is interpreted as a comment, even if the text is otherwise valid C++ syntax. These are sometimes called "C-style" comments, as this comment syntax is inherited from C++'s predecessor language, C:

```
int main()
{
    /*
     *  This is a block comment.
     */
    int a;
}
```

In any block comment, you can write anything you want. When the compiler encounters the symbol `*/`, it terminates the block comment:

```
int main()
{
    /* A block comment with the symbol /*
       Note that the compiler is not affected by the second /*
       however, once the end-block-comment symbol is reached,
       the comment ends.
    */
    int a;
}
```

The above example is valid C++ (and C) code. However, having additional `/*` inside a block comment might result in a warning on some compilers.

Block comments can also start and end *within* a single line. For example:

```
void SomeFunction(/* argument 1 */ int a, /* argument 2 */ int b);
```

Importance of Comments

As with all programming languages, comments provide several benefits:

- Explicit documentation of code to make it easier to read/maintain
- Explanation of the purpose and functionality of code
- Details on the history or reasoning behind the code
- Placement of copyright/licenses, project notes, special thanks, contributor credits, etc. directly in the source code.

However, comments also have their downsides:

- They must be maintained to reflect any changes in the code
- Excessive comments tend to make the code *less* readable

The need for comments can be reduced by writing clear, self-documenting code. A simple example is the use of explanatory names for variables, functions, and types. Factoring out logically related tasks into discrete functions goes hand-in-hand with this.

Comment markers used to disable code

During development, comments can also be used to quickly disable portions of code without deleting it. This is often useful for testing or debugging purposes, but is not good style for anything other than temporary edits. This is often referred to as “commenting out”.

Similarly, keeping old versions of a piece of code in a comment for reference purposes is frowned upon, as it clutters files while offering little value compared to exploring the code's history via a versioning system.

Section 1.3: The standard C++ compilation process

Executable C++ program code is usually produced by a compiler.

A **compiler** is a program that translates code from a programming language into another form which is (more) directly executable for a computer. Using a compiler to translate code is called **compilation.**

C++ inherits the form of its compilation process from its "parent" language, C. Below is a list showing the four major steps of compilation in C++:

1. The C++ preprocessor copies the contents of any included header files into the source code file, generates macro code, and replaces symbolic constants defined using #define with their values.
2. The expanded source code file produced by the C++ preprocessor is compiled into assembly language appropriate for the platform.
3. The assembler code generated by the compiler is assembled into appropriate object code for the platform.
4. The object code file generated by the assembler is linked together with the object code files for any library functions used to produce an executable file.

- Note: some compiled code is linked together, but not to create a final program. Usually, this "linked" code can also be packaged into a format that can be used by other programs. This "bundle of packaged, usable code" is what C++ programmers refer to as a **library.**

Many C++ compilers may also merge or un-merge certain parts of the compilation process for ease or for additional analysis. Many C++ programmers will use different tools, but all of the tools will generally follow this generalized process when they are involved in the production of a program.

The link below extends this discussion and provides a nice graphic to help. [1]:
http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/compile.html

Section 1.4: Function

A **function** is a unit of code that represents a sequence of statements.

Functions can accept **arguments** or values and **return** a single value (or not). To use a function, a **function call** is used on argument values and the use of the function call itself is replaced with its return value.

Every function has a **type signature** -- the types of its arguments and the type of its return type.

Functions are inspired by the concepts of the procedure and the mathematical function.

- Note: C++ functions are essentially procedures and do not follow the exact definition or rules of mathematical functions.

Functions are often meant to perform a specific task. and can be called from other parts of a program. A function must be declared and defined before it is called elsewhere in a program.

- Note: popular function definitions may be hidden in other included files (often for convenience and reuse across many files). This is a common use of header files.

Function Declaration

A **function declaration** is declares the existence of a function with its name and type signature to the compiler. The syntax is as the following:

```
int add2(int i); // The function is of the type (int) -> (int)
```

In the example above, the `int add2(int i)` function declares the following to the compiler:

- The **return type** is `int`.
- The **name** of the function is `add2`.
- The **number of arguments** to the function is 1:
  - The first argument is of the type `int`.
  - The first argument will be referred to in the function's contents by the name `i`.

The argument name is optional; the declaration for the function could also be the following:

```
int add2(int); // Omitting the function arguments' name is also permitted.
```

Per the **one-definition rule**, a function with a certain type signature can only be declared or defined once in an entire C++ code base visible to the C++ compiler. In other words, functions with a specific type signature cannot be re-defined -- they must only be defined once. Thus, the following is not valid C++:

```
int add2(int i);  // The compiler will note that add2 is a function (int) -> int
int add2(int j);  // As add2 already has a definition of (int) -> int, the compiler
                  // will regard this as an error.
```

If a function returns nothing, its return type is written as `void`. If it takes no parameters, the parameter list should be empty.

```
void do_something(); // The function takes no parameters, and does not return anything.
                     // Note that it can still affect variables it has access to.
```

Function Call

A function can be called after it has been declared. For example, the following program calls `add2` with the value of `2` within the function of `main`:

```
#include <iostream>

int add2(int i);    // Declaration of add2

// Note: add2 is still missing a DEFINITION.
// Even though it doesn't appear directly in code,
// add2's definition may be LINKED in from another object file.

int main()
{
    std::cout << add2(2) << "\n";  // add2(2) will be evaluated at this point,
                                   // and the result is printed.
    return 0;
}
```

Here, `add2(2)` is the syntax for a function call.

Function Definition

A *function definition** is similar to a declaration, except it also contains the code that is executed when the function is called within its body.

An example of a function definition for `add2` might be:

```
int add2(int i)       // Data that is passed into (int i) will be referred to by the name i
{                     // while in the function's curly brackets or "scope."

    int j = i + 2;    // Definition of a variable j as the value of i+2.
    return j;         // Returning or, in essence, substitution of j for a function call to
                      // add2.
}
```

Function Overloading

You can create multiple functions with the same name but different parameters.

```
int add2(int i)           // Code contained in this definition will be evaluated
{                         // when add2() is called with one parameter.
    int j = i + 2;
    return j;
}

int add2(int i, int j)    // However, when add2() is called with two parameters, the
{                         // code from the initial declaration will be overloaded,
    int k = i + j + 2 ;   // and the code in this declaration will be evaluated
    return k;             // instead.
}
```

Both functions are called by the same name `add2`, but the actual function that is called depends directly on the amount and type of the parameters in the call. In most cases, the C++ compiler can compute which function to call. In some cases, the type must be explicitly stated.

Default Parameters

Default values for function parameters can only be specified in function declarations.

```
int multiply(int a, int b = 7); // b has default value of 7.
int multiply(int a, int b)
{
    return a * b;               // If multiply() is called with one parameter, the
}                               // value will be multiplied by the default, 7.
```

In this example, `multiply()` can be called with one or two parameters. If only one parameter is given, `b` will have default value of 7. Default arguments must be placed in the latter arguments of the function. For example:

```
int multiply(int a = 10, int b = 20); // This is legal
int multiply(int a = 10, int b);      // This is illegal since int a is in the former
```

Special Function Calls - Operators

There exist special function calls in C++ which have different syntax than `name_of_function(value1, value2, value3)`. The most common example is that of operators.

Certain special character sequences that will be reduced to function calls by the compiler, such as `!`, `+`, `-`, `*`, `%`, and `<<` and many more. These special characters are normally associated with non-programming usage or are used for aesthetics (e.g. the + character is commonly recognized as the addition symbol both within C++ programming as well as in elementary math).

C++ handles these character sequences with a special syntax; but, in essence, each occurrence of an operator is reduced to a function call. For example, the following C++ expression:

```
3+3
```

is equivalent to the following function call:

```
operator+(3, 3)
```

All operator function names start with `operator`.

While in C++'s immediate predecessor, C, operator function names cannot be assigned different meanings by providing additional definitions with different type signatures, in C++, this is valid. "Hiding" additional function definitions under one unique function name is referred to as **operator overloading** in C++, and is a relatively common, but not universal, convention in C++.

Section 1.5: Visibility of function prototypes and declarations

In C++, code must be declared or defined before usage. For example, the following produces a compile time error:

```
int main()
{
  foo(2); // error: foo is called, but has not yet been declared
}

void foo(int x) // this later definition is not known in main
{
}
```

There are two ways to resolve this: putting either the definition or declaration of `foo()` before its usage in `main()`. Here is one example:

```
void foo(int x) {}  //Declare the foo function and body first

int main()
{
  foo(2); // OK: foo is completely defined beforehand, so it can be called here.
}
```

However it is also possible to "forward-declare" the function by putting only a "prototype" declaration before its usage and then defining the function body later:

```
void foo(int);  // Prototype declaration of foo, seen by main
                // Must specify return type, name, and argument list types
int main()
{
  foo(2); // OK: foo is known, called even though its body is not yet defined
}

void foo(int x) //Must match the prototype
{
    // Define body of foo here
}
```

The prototype must specify the return type (`void`), the name of the function (`foo`), and the argument list variable types (`int`), but the names of the arguments are NOT required.

One common way to integrate this into the organization of source files is to make a header file containing all of the prototype declarations:

```
// foo.h
void foo(int); // prototype declaration
```

and then provide the full definition elsewhere:

```
// foo.cpp --> foo.o
#include "foo.h" // foo's prototype declaration is "hidden" in here
void foo(int x) { } // foo's body definition
```

and then, once compiled, link the corresponding object file `foo.o` into the compiled object file where it is used in the linking phase, `main.o`:

```
// main.cpp --> main.o
#include "foo.h" // foo's prototype declaration is "hidden" in here
int main() { foo(2); } // foo is valid to call because its prototype declaration was beforehand.
// the prototype and body definitions of foo are linked through the object files
```

An “unresolved external symbol” error occurs when the function *prototype* and *call* exist, but the function *body* is not defined. These can be trickier to resolve as the compiler won't report the error until the final linking stage, and it doesn't know which line to jump to in the code to show the error.

Section 1.6: Preprocessor

The preprocessor is an important part of the compiler.

It edits the source code, cutting some bits out, changing others, and adding other things.

In source files, we can include preprocessor directives. These directives tells the preprocessor to perform specific actions. A directive starts with a # on a new line. Example:

```
#define ZERO 0
```

The first preprocessor directive you will meet is probably the

```
#include <something>
```

directive. What it does is takes all of `something` and inserts it in your file where the directive was. The hello world program starts with the line

```
#include <iostream>
```

This line adds the functions and objects that let you use the standard input and output.

The C language, which also uses the preprocessor, does not have as many header files as the C++ language, but in C++ you can use all the C header files.

The next important directive is probably the

```
#define something something_else
```

directive. This tells the preprocessor that as it goes along the file, it should replace every occurrence of `something` with `something_else`. It can also make things similar to functions, but that probably counts as advanced C++.

The `something_else` is not needed, but if you define `something` as nothing, then outside preprocessor directives, all occurrences of `something` will vanish.

This actually is useful, because of the `#if`,`#else` and `#ifdef` directives. The format for these would be the following:

```
#if something==true
//code
#else
//more code
#endif

#ifdef thing_that_you_want_to_know_if_is_defined
//code
#endif
```

These directives insert the code that is in the true bit, and deletes the false bits. this can be used to have bits of code that are only included on certain operating systems, without having to rewrite the whole code.

Chapter 2: Literals

Traditionally, a literal is an expression denoting a constant whose type and value are evident from its spelling. For example, `42` is a literal, while `x` is not since one must see its declaration to know its type and read previous lines of code to know its value.

However, C++11 also added user-defined literals, which are not literals in the traditional sense but can be used as a shorthand for function calls.

Section 2.1: this

Within a member function of a class, the keyword `this` is a pointer to the instance of the class on which the function was called. `this` cannot be used in a static member function.

```
struct S {
    int x;
    S& operator=(const S& other) {
        x = other.x;
        // return a reference to the object being assigned to
        return *this;
    }
};
```

The type of `this` depends on the cv-qualification of the member function: if `X::f` is `const`, then the type of `this` within `f` is `const X*`, so `this` cannot be used to modify non-static data members from within a `const` member function. Likewise, `this` inherits `volatile` qualification from the function it appears in.

Version ≥ C++11

`this` can also be used in a *brace-or-equal-initializer* for a non-static data member.

```
struct S;
struct T {
    T(const S* s);
    // ...
};
struct S {
    // ...
    T t{this};
};
```

`this` is an rvalue, so it cannot be assigned to.

Section 2.2: Integer literal

An integer literal is a primary expression of the form

- decimal-literal

It is a non-zero decimal digit (1, 2, 3, 4, 5, 6, 7, 8, 9), followed by zero or more decimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

```
int d = 42;
```

- octal-literal

It is the digit zero (0) followed by zero or more octal digits (0, 1, 2, 3, 4, 5, 6, 7)

```
int o = 052
```

- hex-literal

It is the character sequence 0x or the character sequence 0X followed by one or more hexadecimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C, d, D, e, E, f, F)

```
int x = 0x2a; int X = 0X2A;
```

- binary-literal (since C++14)

It is the character sequence 0b or the character sequence 0B followed by one or more binary digits (0, 1)

```
int b = 0b101010; // C++14
```

Integer-suffix, if provided, may contain one or both of the following (if both are provided, they may appear in any order:

- unsigned-suffix (the character u or the character U)

```
unsigned int u_1 = 42u;
```

- long-suffix (the character l or the character L) or the long-long-suffix (the character sequence ll or the character sequence LL) (since C++11)

The following variables are also initialized to the same value:

```
unsigned long long l1 = 18446744073709550592ull; // C++11
unsigned long long l2 = 18'446'744'073'709'550'592llu; // C++14
unsigned long long l3 = 1844'6744'0737'0955'0592uLL; // C++14
unsigned long long l4 = 184467'440737'0'95505'92LLU; // C++14
```

Notes

Letters in the integer literals are case-insensitive: 0xDeAdBaBeU and 0XdeadBABEu represent the same number (one exception is the long-long-suffix, which is either ll or LL, never lL or Ll)

There are no negative integer literals. Expressions such as -1 apply the unary minus operator to the value represented by the literal, which may involve implicit type conversions.

In C prior to C99 (but not in C++), unsuffixed decimal values that do not fit in long int are allowed to have the type unsigned long int.

When used in a controlling expression of #if or #elif, all signed integer constants act as if they have type std::intmax_t and all unsigned integer constants act as if they have type std::uintmax_t.

Section 2.3: true

A keyword denoting one of the two possible values of type `bool`.

```
bool ok = true;
if (!f()) {
    ok = false;
    goto end;
}
```

Section 2.4: false

A keyword denoting one of the two possible values of type `bool`.

```
bool ok = true;
if (!f()) {
    ok = false;
    goto end;
}
```

Section 2.5: nullptr

Version ≥ C++11

A keyword denoting a null pointer constant. It can be converted to any pointer or pointer-to-member type, yielding a null pointer of the resulting type.

```
Widget* p = new Widget();
delete p;
p = nullptr; // set the pointer to null after deletion
```

Note that `nullptr` is not itself a pointer. The type of `nullptr` is a fundamental type known as `std::nullptr_t`.

```
void f(int* p);

template <class T>
void g(T* p);

void h(std::nullptr_t p);

int main() {
    f(nullptr); // ok
    g(nullptr); // error
    h(nullptr); // ok
}
```

Chapter 3: operator precedence

Section 3.1: Logical && and || operators: short-circuit

&& has precedence over ||, this means that parentheses are placed to evaluate what would be evaluated together.

c++ uses short-circuit evaluation in && and || to not do unnecessary executions.
If the left hand side of || returns true the right hand side does not need to be evaluated anymore.

```
#include <iostream>
#include <string>

using namespace std;

bool True(string id){
    cout << "True" << id << endl;
    return true;
}

bool False(string id){
    cout << "False" << id << endl;
    return false;
}


int main(){
    bool result;
    //let's evaluate 3 booleans with || and && to illustrate operator precedence
    //precedence does not mean that && will be evaluated first but rather where
    //parentheses would be added
    //example 1
    result =
        False("A") || False("B") && False("C");
                // eq. False("A") || (False("B") && False("C"))
    //FalseA
    //FalseB
    //"Short-circuit evaluation skip of C"
    //A is false so we have to evaluate the right of ||,
    //B being false we do not have to evaluate C to know that the result is false



    result =
        True("A") || False("B") && False("C");
                // eq. True("A") || (False("B") && False("C"))
    cout << result << " :=====================" << endl;
    //TrueA
    //"Short-circuit evaluation skip of B"
    //"Short-circuit evaluation skip of C"
    //A is true so we do not have to evaluate
    //        the right of || to know that the result is true
    //If || had precedence over && the equivalent evaluation would be:
    // (True("A") || False("B")) && False("C")
    //What would print
    //TrueA
    //"Short-circuit evaluation skip of B"
    //FalseC
    //Because the parentheses are placed differently
    //the parts that get evaluated are differently
    //which makes that the end result in this case would be False because C is false
```

```
}
```

Section 3.2: Unary Operators

Unary operators act on the object upon which they are called and have high precedence. (See Remarks)

When used postfix, the action occurs only after the entire operation is evaluated, leading to some interesting arithmetics:

```
int a = 1;
++a;            // result: 2
a--;            // result: 1
int minusa=-a;  // result: -1

bool b = true;
!b; // result: true

a=4;
int c = a++/2;      // equal to: (a==4) 4 / 2   result: 2 ('a' incremented postfix)
cout << a << endl;  // prints 5!
int d = ++a/2;      // equal to: (a+1) == 6 / 2 result: 3

int arr[4] =  {1,2,3,4};

int *ptr1 = &arr[0];    // points to arr[0] which is 1
int *ptr2 = ptr1++;     // ptr2 points to arr[0] which is still 1; ptr1 incremented
std::cout << *ptr1++ << std::endl;  // prints  2

int e = arr[0]++;       // receives the value of arr[0] before it is incremented
std::cout << e << std::endl;      // prints 1
std::cout << *ptr2 << std::endl;  // prints arr[0] which is now 2
```

Section 3.3: Arithmetic operators

Arithmetic operators in C++ have the same precedence as they do in mathematics:

Multiplication and division have left associativity(meaning that they will be evaluated from left to right) and they have higher precedence than addition and subtraction, which also have left associativity.

We can also force the precedence of expression using parentheses `( )`. Just the same way as you would do that in normal mathematics.

```
// volume of a spherical shell = 4 pi R^3 - 4 pi r^3
double vol = 4.0*pi*R*R*R/3.0 - 4.0*pi*r*r*r/3.0;

//Addition:

int a = 2+4/2;          // equal to: 2+(4/2)         result: 4
int b = (3+3)/2;        // equal to: (3+3)/2         result: 3

//With Multiplication

int c = 3+4/2*6;        // equal to: 3+((4/2)*6)     result: 15
int d = 3*(3+6)/9;      // equal to: (3*(3+6))/9     result: 3

//Division and Modulo

int g = 3-3%1;          // equal to: 3 % 1 = 0  3 - 0 = 3
int h = 3-(3%1);        // equal to: 3 % 1 = 0  3 - 0 = 3
```

```
int i = 3-3/1%3;        // equal to: 3 / 1 = 3  3 % 3 = 0  3 - 0 = 3
int l = 3-(3/1)%3;      // equal to: 3 / 1 = 3  3 % 3 = 0  3 - 0 = 3
int m = 3-(3/(1%3));    // equal to: 1 % 3 = 1  3 / 1 = 3  3 - 3 = 0
```

Section 3.4: Logical AND and OR operators

These operators have the usual precedence in C++: AND before OR.

```
// You can drive with a foreign license for up to 60 days
bool can_drive = has_domestic_license || has_foreign_license && num_days <= 60;
```

This code is equivalent to the following:

```
// You can drive with a foreign license for up to 60 days
bool can_drive = has_domestic_license || (has_foreign_license && num_days <= 60);
```

Adding the parenthesis does not change the behavior, though, it does make it easier to read. By adding these parentheses, no confusion exist about the intent of the writer.

Chapter 4: Floating Point Arithmetic

Section 4.1: Floating Point Numbers are Weird

The first mistake that nearly every single programmer makes is presuming that this code will work as intended:

```
float total = 0;
for(float a = 0; a != 2; a += 0.01f) {
    total += a;
}
```

The novice programmer assumes that this will sum up every single number in the range `0, 0.01, 0.02, 0.03, ..., 1.97, 1.98, 1.99`, to yield the result `199`—the mathematically correct answer.

Two things happen that make this untrue:

1. The program as written never concludes. `a` never becomes equal to `2`, and the loop never terminates.
2. If we rewrite the loop logic to check `a < 2` instead, the loop terminates, but the total ends up being something different from `199`. On IEEE754-compliant machines, it will often sum up to about `201` instead.

The reason that this happens is that **Floating Point Numbers represent Approximations of their assigned values**.

The classical example is the following computation:

```
double a = 0.1;
double b = 0.2;
double c = 0.3;
if(a + b == c)
    //This never prints on IEEE754-compliant machines
    std::cout << "This Computer is Magic!" << std::endl;
else
    std::cout << "This Computer is pretty normal, all things considered." << std::endl;
```

Though what we the programmer see is three numbers written in base10, what the compiler (and the underlying hardware) see are binary numbers. Because `0.1`, `0.2`, and `0.3` require perfect division by `10`—which is quite easy in a base-10 system, but impossible in a base-2 system—these numbers have to be stored in imprecise formats, similar to how the number `1/3` has to be stored in the imprecise form `0.333333333333333...` in base-10.

```
//64-bit floats have 53 digits of precision, including the whole-number-part.
double a =     0011111110111001100110011001100110011001100110011001100110011010; //imperfect representation of 0.1
double b =     0011111111001001100110011001100110011001100110011001100110011010; //imperfect representation of 0.2
double c =     0011111111010011001100110011001100110011001100110011001100110011; //imperfect representation of 0.3
double a + b = 0011111111010011001100110011001100110011001100110011001100110100; //Note that this is not quite equal to the "canonical" 0.3!
```

Chapter 5: Bit Operators

Section 5.1: | - bitwise OR

```
int a = 5;     // 0101b  (0x05)
int b = 12;    // 1100b  (0x0C)
int c = a | b; // 1101b  (0x0D)

std::cout << "a = " << a << ", b = " << b << ", c = " << c << std::endl;
```

Output

```
a = 5, b = 12, c = 13
```

Why

A bit wise `OR` operates on the bit level and uses the following Boolean truth table:

```
true OR true = true
true OR false = true
false OR false = false
```

When the binary value for `a` (`0101`) and the binary value for `b` (`1100`) are `OR`'ed together we get the binary value of `1101`:

```
int a = 0 1 0 1
int b = 1 1 0 0 |
        ---------
int c = 1 1 0 1
```

The bit wise OR does not change the value of the original values unless specifically assigned to using the bit wise assignment compound operator `|=`:

```
int a = 5;  // 0101b  (0x05)
a |= 12;    // a = 0101b | 1101b
```

Section 5.2: ^ - bitwise XOR (exclusive OR)

```
int a = 5;     // 0101b  (0x05)
int b = 9;     // 1001b  (0x09)
int c = a ^ b; // 1100b  (0x0C)

std::cout << "a = " << a << ", b = " << b << ", c = " << c << std::endl;
```

Output

```
a = 5, b = 9, c = 12
```

Why

A bit wise `XOR` (exclusive or) operates on the bit level and uses the following Boolean truth table:

```
true OR true = false
true OR false = true
false OR false = false
```

Notice that with an XOR operation `true OR true = false` where as with operations `true AND/OR true = true`, hence the exclusive nature of the XOR operation.

Using this, when the binary value for `a` (`0101`) and the binary value for `b` (`1001`) are `XOR`'ed together we get the binary value of `1100`:

```
int a = 0 1 0 1
int b = 1 0 0 1 ^
        ---------
int c = 1 1 0 0
```

The bit wise XOR does not change the value of the original values unless specifically assigned to using the bit wise assignment compound operator `^=`:

```
int a = 5;  // 0101b  (0x05)
a ^= 9;    // a = 0101b ^ 1001b
```

The bit wise XOR can be utilized in many ways and is often utilized in bit mask operations for encryption and compression.

Note: The following example is often shown as an example of a nice trick. But should not be used in production code (there are better ways `std::swap()` to achieve the same result).

You can also utilize an XOR operation to swap two variables without a temporary:

```
int a = 42;
int b = 64;

// XOR swap
a ^= b;
b ^= a;
a ^= b;

std::cout << "a = " << a << ", b = " << b << "\n";
```

To productionalize this you need to add a check to make sure it can be used.

```
void doXORSwap(int& a, int& b)
{
    // Need to add a check to make sure you are not swapping the same
    // variable with itself. Otherwise it will zero the value.
    if (&a != &b)
    {
        // XOR swap
        a ^= b;
        b ^= a;
        a ^= b;
    }
}
```

So though it looks like a nice trick in isolation it is not useful in real code. xor is not a base logical operation,but a combination of others: a^c=~(a&c)&(a|c)

also in 2015+ compilers variables may be assigned as binary:

```
int cn=0b0111;
```

Section 5.3: & - bitwise AND

```
int a = 6;     // 0110b  (0x06)
int b = 10;    // 1010b  (0x0A)
int c = a & b; // 0010b  (0x02)

std::cout << "a = " << a << ", b = " << b << ", c = " << c << std::endl;
```

Output

`a = 6, b = 10, c = 2`

Why

A bit wise `AND` operates on the bit level and uses the following Boolean truth table:

```
TRUE  AND TRUE  = TRUE
TRUE  AND FALSE = FALSE
FALSE AND FALSE = FALSE
```

When the binary value for `a` (`0110`) and the binary value for `b` (`1010`) are `AND`'ed together we get the binary value of `0010`:

```
int a = 0 1 1 0
int b = 1 0 1 0 &
        ---------
int c = 0 0 1 0
```

The bit wise AND does not change the value of the original values unless specifically assigned to using the bit wise assignment compound operator `&=`:

```
int a = 5;  // 0101b  (0x05)
a &= 10;    // a = 0101b & 1010b
```

Section 5.4: << - left shift

```
int a = 1;      // 0001b
int b = a << 1; // 0010b

std::cout << "a = " << a << ", b = " << b << std::endl;
```

Output

`a = 1, b = 2`

Why

The left bit wise shift will shift the bits of the left hand value (`a`) the number specified on the right (`1`), essentially padding the least significant bits with 0's, so shifting the value of `5` (binary `0000 0101`) to the left 4 times (e.g. `5 << 4`) will yield the value of `80` (binary `0101 0000`). You might note that shifting a value to the left 1 time is also the same as multiplying the value by 2, example:

```
int a = 7;
while (a < 200) {
    std::cout << "a = " << a << std::endl;
    a <<= 1;
```

```
}

a = 7;
while (a < 200) {
    std::cout << "a = " << a << std::endl;
    a *= 2;
}
```

But it should be noted that the left shift operation will shift *all* bits to the left, including the sign bit, example:

```
int a = 2147483647; // 0111 1111 1111 1111 1111 1111 1111 1111
int b = a << 1;     // 1111 1111 1111 1111 1111 1111 1111 1110

std::cout << "a = " << a << ", b = " << b << std::endl;
```

Possible output: `a = 2147483647, b = -2`

While some compilers will yield results that seem expected, it should be noted that if you left shift a signed number so that the sign bit is affected, the result is **undefined**. It is also **undefined** if the number of bits you wish to shift by is a negative number or is larger than the number of bits the type on the left can hold, example:

```
int a = 1;
int b = a << -1;  // undefined behavior
char c = a << 20; // undefined behavior
```

The bit wise left shift does not change the value of the original values unless specifically assigned to using the bit wise assignment compound operator `<<=`:

```
int a = 5;  // 0101b
a <<= 1;    // a = a << 1;
```

Section 5.5: >> - right shift

```
int a = 2;      // 0010b
int b = a >> 1; // 0001b

std::cout << "a = " << a << ", b = " << b << std::endl;
```

Output

`a = 2, b = 1`

Why

The right bit wise shift will shift the bits of the left hand value (a) the number specified on the right (1); it should be noted that while the operation of a right shift is standard, what happens to the bits of a right shift on a *signed negative* number is *implementation defined* and thus cannot be guaranteed to be portable, example:

```
int a = -2;
int b = a >> 1; // the value of b will be depend on the compiler
```

It is also undefined if the number of bits you wish to shift by is a negative number, example:

```
int a = 1;
int b = a >> -1;  // undefined behavior
```

The bit wise right shift does not change the value of the original values unless specifically assigned to using the bit wise assignment compound operator `>>=`:

```
int a = 2;  // 0010b
a >>= 1;    // a = a >> 1;
```

Chapter 6: Bit Manipulation

Section 6.1: Remove rightmost set bit

C-style bit-manipulation

```
template <typename T>
T rightmostSetBitRemoved(T n)
{
    // static_assert(std::is_integral<T>::value && !std::is_signed<T>::value, "type should be
unsigned"); // For c++11 and later
    return n & (n - 1);
}
```

Explanation

- if `n` is zero, we have `0 & 0xFF..FF` which is zero
- else `n` can be written `0bxxxxxx10..00` and `n - 1` is `0bxxxxxx011..11`, so `n & (n - 1)` is `0bxxxxxx000..00`.

Section 6.2: Set all bits

C-style bit-manipulation

```
x = -1; // -1 == 1111 1111 ... 1111b
```

(See here for an explanation of why this works and is actually the best approach.)

Using std::bitset

```
std::bitset<10> x;
x.set(); // Sets all bits to '1'
```

Section 6.3: Toggling a bit

C-style bit-manipulation

A bit can be toggled using the XOR operator (`^`).

```
// Bit x will be the opposite value of what it is currently
number ^= 1LL << x;
```

Using std::bitset

```
std::bitset<4> num(std::string("0100"));
num.flip(2); // num is now 0000
num.flip(0); // num is now 0001
num.flip();  // num is now 1110 (flips all bits)
```

Section 6.4: Checking a bit

C-style bit-manipulation

The value of the bit can be obtained by shifting the number to the right `x` times and then performing bitwise AND (`&`) on it:

```
(number >> x) & 1LL;  // 1 if the 'x'th bit of 'number' is set, 0 otherwise
```

The right-shift operation may be implemented as either an arithmetic (signed) shift or a logical (unsigned) shift. If `number` in the expression `number >> x` has a signed type and a negative value, the resulting value is implementation-defined.

If we need the value of that bit directly in-place, we could instead left shift the mask:

```
(number & (1LL << x));  // (1 << x) if the 'x'th bit of 'number' is set, 0 otherwise
```

Either can be used as a conditional, since all non-zero values are considered true.

Using std::bitset

```
std::bitset<4> num(std::string("0010"));
bool bit_val = num.test(1);  // bit_val value is set to true;
```

Section 6.5: Counting bits set

The population count of a bitstring is often needed in cryptography and other applications and the problem has been widely studied.

The naive way requires one iteration per bit:

```
unsigned value = 1234;
unsigned bits = 0;  // accumulates the total number of bits set in `n`

for (bits = 0; value; value >>= 1)
  bits += value & 1;
```

A nice trick (based on Remove rightmost set bit) is:

```
unsigned bits = 0;  // accumulates the total number of bits set in `n`

for (; value; ++bits)
  value &= value - 1;
```

It goes through as many iterations as there are set bits, so it's good when `value` is expected to have few nonzero bits.

The method was first proposed by Peter Wegner (in CACM 3 / 322 - 1960) and it's well known since it appears in *C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie.

This requires 12 arithmetic operations, one of which is a multication:

```
unsigned popcount(std::uint64_t x)
{
  const std::uint64_t m1  = 0x5555555555555555;  // binary: 0101...
  const std::uint64_t m2  = 0x3333333333333333;  // binary: 00110011..
  const std::uint64_t m4  = 0x0f0f0f0f0f0f0f0f;  // binary: 0000111100001111

  x -= (x >> 1) & m1;             // put count of each 2 bits into those 2 bits
  x = (x & m2) + ((x >> 2) & m2); // put count of each 4 bits into those 4 bits
  x = (x + (x >> 4)) & m4;        // put count of each 8 bits into those 8 bits
  return (x * h01) >> 56;  // left 8 bits of x + (x<<8) + (x<<16) + (x<<24) + ...
}
```

This kind of implementation has the best worst-case behavior (see Hamming weight for further details).

Many CPUs have a specific instruction (like x86's `popcnt`) and the compiler could offer a specific (**non standard**) built in function. E.g. with g++ there is:

```
int __builtin_popcount (unsigned x);
```

Section 6.6: Check if an integer is a power of 2

The `n & (n - 1)` trick (see Remove rightmost set bit) is also useful to determine if an integer is a power of 2:

```
bool power_of_2 = n && !(n & (n - 1));
```

Note that without the first part of the check (`n &&`), `0` is incorrectly considered a power of 2.

Section 6.7: Setting a bit

C-style bit manipulation

A bit can be set using the bitwise OR operator (`|`).

```
// Bit x will be set
number |= 1LL << x;
```

Using std::bitset

`set(x)` or `set(x,true)` - sets bit at position `x` to `1`.

```
std::bitset<5> num(std::string("01100"));
num.set(0);      // num is now 01101
num.set(2);      // num is still 01101
num.set(4,true); // num is now 11110
```

Section 6.8: Clearing a bit

C-style bit-manipulation

A bit can be cleared using the bitwise AND operator (`&`).

```
// Bit x will be cleared
number &= ~(1LL << x);
```

Using std::bitset

`reset(x)` or `set(x,false)` - clears the bit at position `x`.

```
std::bitset<5> num(std::string("01100"));
num.reset(2);     // num is now 01000
num.reset(0);     // num is still 01000
num.set(3,false); // num is now 00000
```

Section 6.9: Changing the nth bit to x

C-style bit-manipulation

```
// Bit n will be set if x is 1 and cleared if x is 0.
number ^= (-x ^ number) & (1LL << n);
```

Using std::bitset

`set(n,val)` - sets bit `n` to the value `val`.

```
std::bitset<5> num(std::string("00100"));
num.set(0,true);  // num is now 00101
num.set(2,false); // num is now 00001
```

Section 6.10: Bit Manipulation Application: Small to Capital Letter

One of several applications of bit manipulation is converting a letter from small to capital or vice versa by choosing a **mask** and a proper **bit operation**. For example, the **a** letter has this binary representation `01(1)00001` while its capital counterpart has `01(0)00001`. They differ solely in the bit in parenthesis. In this case, converting the **a** letter from small to capital is basically setting the bit in parenthesis to one. To do so, we do the following:

```
/****************************************
convert small letter to captial letter.
========================================
     a: 01100001
  mask: 11011111  <-- (0xDF)  11(0)11111
      :---------
a&mask: 01000001  <-- A letter
*****************************************/
```

The code for converting a letter to A letter is

```
#include <cstdio>

int main()
{
    char op1 = 'a';  // "a" letter (i.e. small case)
    int mask = 0xDF; // choosing a proper mask

    printf("a (AND) mask = A\n");
    printf("%c   &   0xDF = %c\n", op1, op1 & mask);

    return 0;
}
```

The result is

```
$ g++ main.cpp -o test1
$ ./test1
a (AND) mask = A
a   &   0xDF = A
```

Chapter 7: Bit fields

Bit fields tightly pack C and C++ structures to reduce size. This appears painless: specify the number of bits for members, and compiler does the work of co-mingling bits. The restriction is inability to take the address of a bit field member, since it is stored co-mingled. `sizeof()` is also disallowed.

The cost of bit fields is slower access, as memory must be retrieved and bitwise operations applied to extract or modify member values. These operations also add to executable size.

Section 7.1: Declaration and Usage

```
struct FileAttributes
{
    unsigned int ReadOnly: 1;
    unsigned int Hidden: 1;
};
```

Here, each of these two fields will occupy 1 bit in memory. It is specified by `: 1` expression after the variable names. Base type of bit field could be any integral type (8-bit int to 64-bit int). Using `unsigned` type is recommended, otherwise surprises may come.

If more bits are required, replace "1" with number of bits required. For example:

```
struct Date
{
    unsigned int Year : 13; // 2^13 = 8192, enough for "year" representation for long time
    unsigned int Month: 4;  // 2^4 = 16, enough to represent 1-12 month values.
    unsigned int Day:   5;  // 32
};
```

The whole structure is using just 22 bits, and with normal compiler settings, `sizeof` this structure would be 4 bytes.

Usage is pretty simple. Just declare the variable, and use it like ordinary structure.

```
Date d;

d.Year = 2016;
d.Month = 7;
d.Day =  22;

std::cout << "Year:" << d.Year << std::endl <<
        "Month:" << d.Month << std::endl <<
        "Day:" << d.Day << std::endl;
```

Chapter 8: Arrays

Arrays are elements of the same type placed in adjoining memory locations. The elements can be individually referenced by a unique identifier with an added index.

This allows you to declare multiple variable values of a specific type and access them individually without needing to declare a variable for each value.

Section 8.1: Array initialization

An array is just a block of sequential memory locations for a specific type of variable. Arrays are allocated the same way as normal variables, but with square brackets appended to its name `[]` that contain the number of elements that fit into the array memory.

The following example of an array uses the typ `int`, the variable name `arrayOfInts`, and the number of elements `[5]` that the array has space for:

```
int arrayOfInts[5];
```

An array can be declared and initialized at the same time like this

```
int arrayOfInts[5] = {10, 20, 30, 40, 50};
```

When initializing an array by listing all of its members, it is not necessary to include the number of elements inside the square brackets. It will be automatically calculated by the compiler. In the following example, it's 5:

```
int arrayOfInts[] = {10, 20, 30, 40, 50};
```

It is also possible to initialize only the first elements while allocating more space. In this case, defining the length in brackets is mandatory. The following will allocate an array of length 5 with partial initialization, the compiler initializes all remaining elements with the standard value of the element type, in this case zero.

```
int arrayOfInts[5] = {10,20}; // means 10, 20, 0, 0, 0
```

Arrays of other basic data types may be initialized in the same way.

```
char arrayOfChars[5]; // declare the array and allocate the memory, don't initialize

char arrayOfChars[5] = { 'a', 'b', 'c', 'd', 'e' } ; //declare and initialize

double arrayOfDoubles[5] = {1.14159, 2.14159, 3.14159, 4.14159, 5.14159};

string arrayOfStrings[5] = { "C++", "is", "super", "duper", "great!"};
```

It is also important to take note that when accessing array elements, the array's element index(or position) starts from 0.

```
int array[5] = { 10/*Element no.0*/, 20/*Element no.1*/, 30, 40, 50/*Element no.4*/};
std::cout << array[4]; //outputs 50
std::cout << array[0]; //outputs 10
```

Section 8.2: A fixed size raw array matrix (that is, a 2D raw array)

```
// A fixed size raw array matrix (that is, a 2D raw array).
#include <iostream>
#include <iomanip>
using namespace std;

auto main() -> int
{
    int const   n_rows  = 3;
    int const   n_cols  = 7;
    int const   m[n_rows][n_cols] =             // A raw array matrix.
    {
        {  1,  2,  3,  4,  5,  6,  7 },
        {  8,  9, 10, 11, 12, 13, 14 },
        { 15, 16, 17, 18, 19, 20, 21 }
    };

    for( int y = 0; y < n_rows; ++y )
    {
        for( int x = 0; x < n_cols; ++x )
        {
            cout << setw( 4 ) << m[y][x];       // Note: do NOT use m[y,x]!
        }
        cout << '\n';
    }
}
```

Output:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

C++ doesn't support special syntax for indexing a multi-dimensional array. Instead such an array is viewed as an array of arrays (possibly of arrays, and so on), and the ordinary single index notation `[i]` is used for each level. In the example above `m[y]` refers to row `y` of `m`, where `y` is a zero-based index. Then this row can be indexed in turn, e.g. `m[y][x]`, which refers to the `x`th item – or column – of row `y`.

I.e. the last index varies fastest, and in the declaration the range of this index, which here is the number of columns per row, is the last and “innermost” size specified.

Since C++ doesn't provide built-in support for dynamic size arrays, other than dynamic allocation, a dynamic size matrix is often implemented as a class. Then the raw array matrix indexing notation `m[y][x]` has some cost, either by exposing the implementation (so that e.g. a view of a transposed matrix becomes practically impossible) or by adding some overhead and slight inconvenience when it's done by returning a proxy object from `operator[]`. And so the indexing notation for such an abstraction can and will usually be different, both in look-and-feel and in the order of indices, e.g. `m(x,y)` or `m.at(x,y)` or `m.item(x,y)`.

Section 8.3: Dynamically sized raw array

```
// Example of raw dynamic size array. It's generally better to use std::vector.
#include <algorithm>            // std::sort
#include <iostream>
using namespace std;

auto int_from( istream& in ) -> int { int x; in >> x; return x; }
```

```
auto main()
    -> int
{
    cout << "Sorting n integers provided by you.\\n";
    cout << "n? ";
    int const   n   = int_from( cin );
    int*        a   = new int[n];       // ← Allocation of array of n items.

    for( int i = 1; i <= n; ++i )
    {
        cout << "The #" << i << " number, please: ";
        a[i-1] = int_from( cin );
    }

    sort( a, a + n );
    for( int i = 0; i < n; ++i ) { cout << a[i] << ' '; }
    cout << '\\n';

    delete[] a;
}
```

A program that declares an array `T a[n];` where `n` is determined a run-time, can compile with certain compilers that support C99 *variadic length arrays* (VLAs) as a language extension. But VLAs are not supported by standard C++. This example shows how to manually allocate a dynamic size array via a `new[]`-expression,

```
int*        a   = new int[n];       // ← Allocation of array of n items.
```

… then use it, and finally deallocate it via a `delete[]`-expression:

```
delete[] a;
```

The array allocated here has indeterminate values, but it can be zero-initialized by just adding an empty parenthesis `()`, like this: `new int[n]()`. More generally, for arbitrary item type, this performs a *value-initialization*.

As part of a function down in a call hierarchy this code would not be exception safe, since an exception before the `delete[]` expression (and after the `new[]`) would cause a memory leak. One way to address that issue is to automate the cleanup via e.g. a `std::unique_ptr` smart pointer. But a generally better way to address it is to just use a `std::vector`: that's what `std::vector` is there for.

Section 8.4: Array size: type safe at compile time

```
#include      // size_t, ptrdiff_t

//------------------------------------ Machinery:

using Size = ptrdiff_t;

template< class Item, size_t n >
constexpr auto n_items( Item (&)[n] ) noexcept
-> Size
{ return n; }


//------------------------------------ Usage:

#include
using namespace std;
auto main()
```

```
-> int
{
int const   a[]     = {3, 1, 4, 1, 5, 9, 2, 6, 5, 4};
Size const  n       = n_items( a );
int         b[n]    = {};       // An array of the same size as a.

(void) b;
cout <}
```

The C idiom for array size, `sizeof(a)/sizeof(a[0])`, will accept a pointer as argument and will then generally yield an incorrect result.

For C++11

using C++11 you can do:

```
std::extent<decltype(MyArray)>::value;
```

Example:

```
char MyArray[] = { 'X','o','c','e' };
const auto n = std::extent<decltype(MyArray)>::value;
std::cout << n << "\n"; // Prints 4
```

Up till C++17 (forthcoming as of this writing) C++ had no built-in core language or standard library utility to obtain the size of an array, but this can be implemented by passing the array *by reference* to a function template, as shown above. Fine but important point: the template size parameter is a `size_t`, somewhat inconsistent with the signed `Size` function result type, in order to accommodate the g++ compiler which sometimes insists on `size_t` for template matching.

With C++17 and later one may instead use `std::size`, which is specialized for arrays.

Section 8.5: Expanding dynamic size array by using std::vector

```
// Example of std::vector as an expanding dynamic size array.
#include <algorithm>            // std::sort
#include <iostream>
#include <vector>               // std::vector
using namespace std;

int int_from( std::istream& in ) { int x = 0; in >> x; return x; }

int main()
{
    cout << "Sorting integers provided by you.\n";
    cout << "You can indicate EOF via F6 in Windows or Ctrl+D in Unix-land.\n";
    vector<int> a;      // ← Zero size by default.

    while( cin )
    {
        cout << "One number, please, or indicate EOF: ";
        int const x = int_from( cin );
        if( !cin.fail() ) { a.push_back( x ); }  // Expands as necessary.
    }

    sort( a.begin(), a.end() );
```

```
    int const n = a.size();
    for( int i = 0; i < n; ++i ) { cout << a[i] << ' '; }
    cout << '\n';
}
```

`std::vector` is a standard library class template that provides the notion of a variable size array. It takes care of all the memory management, and the buffer is contiguous so a pointer to the buffer (e.g. `&v[0]` or `v.data()`) can be passed to API functions requiring a raw array. A `vector` can even be expanded at run time, via e.g. the `push_back` member function that appends an item.

The complexity of the sequence of *n* `push_back` operations, including the copying or moving involved in the vector expansions, is amortized O(*n*). "Amortized": on average.

Internally this is usually achieved by the vector *doubling* its buffer size, its capacity, when a larger buffer is needed. E.g. for a buffer starting out as size 1, and being repeatedly doubled as needed for *n*=17 `push_back` calls, this involves 1 + 2 + 4 + 8 + 16 = 31 copy operations, which is less than 2×*n* = 34. And more generally the sum of this sequence can't exceed 2×*n*.

Compared to the dynamic size raw array example, this `vector`-based code does not require the user to supply (and know) the number of items up front. Instead the vector is just expanded as necessary, for each new item value specified by the user.

Section 8.6: A dynamic size matrix using std::vector for storage

Unfortunately as of C++14 there's no dynamic size matrix class in the C++ standard library. Matrix classes that support dynamic size are however available from a number of 3rd party libraries, including the Boost Matrix library (a sub-library within the Boost library).

If you don't want a dependency on Boost or some other library, then one poor man's dynamic size matrix in C++ is just like

```
vector<vector<int>> m( 3, vector<int>( 7 ) );
```

… where `vector` is `std::vector`. The matrix is here created by copying a row vector *n* times where *n* is the number of rows, here 3. It has the advantage of providing the same `m[y][x]` indexing notation as for a fixed size raw array matrix, but it's a bit inefficient because it involves a dynamic allocation for each row, and it's a bit unsafe because it's possible to inadvertently resize a row.

A more safe and efficient approach is to use a single vector as *storage* for the matrix, and map the client code's (*x*, *y*) to a corresponding index in that vector:

```
// A dynamic size matrix using std::vector for storage.

//--------------------------------------------- Machinery:
#include         // std::copy
#include          // assert
#include  // std::initializer_list
#include            // std::vector
#include          // ptrdiff_t

namespace my {
using Size = ptrdiff_t;
using std::initializer_list;
using std::vector;
```