Yh Akademin

C++ Language

Fundamentals

Preprocessor Directives

🔗 [Preprocessor]

- The preprocessor prepares the actual code for compilation.
- Preprocessing is done in the first step of build
- Preprocessor directives control the behavior of the preprocessor.
- Preprocessor directives are used to do different things
  - Inserting content of header files
  - Defining text macros
  - Conditional compilation
  - Generating errors
  - Controlling line number and filename
- A preprocessor directive begins with **#** ands end with the first **newline**. A multiline can be created using **backslashes** (\)

```
#include <cstdio>

#ifndef BUFFER_SIZE
#error "The buffer size shall be defined"
#endif

#define DEBUG

#define PI 3.1415

#define SQR(x) ((x) * (x))

int main(void) {
#ifdef DEBUG
    printf("A debug message");
#endif
    return 0;
}
```

Preprocessor Directives - Inclusion

- **#include** is used to insert content of header files. E.g. **#include <iostream>**
- There are two ways to specify header file names:
  - Using angle brackets for standard header files. E.g. **#include <filename>**
  - Using double quotation marks for your own header files. E.g. **#include "filename"**
- The compiler shall be able to resolve path to included files
  - It automatically resolves the path to standard header files
  - For your own header files, you need to help the compiler to resolve paths
    - Relative or absolute paths can be used. E.g. **#include "./lib/mylib.h"** in main.cpp
    - Include only files and specify **directory paths** when we compile source files in command line using **-I.** For example:
      - **#include "mylib.h"** in main.cpp
      - Compile main.cpp using: g++ -c main.cpp **-I./lib** -o main.o

Preprocessor Directives - Macros

- A macro is a fragment of code which has been given a name.
- A macro can be defined using the **#define** preprocessor directive.
  - Object-like macros: **#define macro_name replacement_text**. E.g. **#define PI 3.1415**
    - A common use of object-like macros is to create constants.
      - Avoid using magic numbers and literals in your code
    - There are some predefined macros like **__LINE__**, **__func__**, **__DATE__** and etc.
  - Function-like macros can be defined with or without parameters
    - **#define macro_name([parameter_list]) replacement_text**. E.g. **#define ADD(x,y) ((x)+(y))**
    - Macros with variable number of parameters using spread (…) operator.
      - **#define macro_name([parameter_list ,] ...) replacement_text**.
      - … means one or more parameters. __VA_ARGS__ identifier represents the arguments
        - E.g. #define PRINT(fmt, ...) printf(fmt, __VA_ARGS__)

Preprocessor Directives - Macros

- A macro can be used in another macro. E.g. **#define PI 3.1415** and **#define AREA(r) (PI * (r) *(r))**
- A function-like macro can be called like a function. E.g. int value = ADD(2, 3);
- In function-like macros we shall enclose the parameters using parentheses
- Scope of a macro is file and can be used in another macro before definition.
- Macros can be defined in command line using **-D**. For example:
  - g++ main.cpp **-D**DEBUG or g++ main.cpp **-D**NUM=10 or g++ main.cpp **-D**NAME=\"Eva\"
- **#undef** can be used to undefine a macro. E.g. #undef PI
- To undefine macros not defined in your source code in command line use **-U**.
  - E.g. g++ main.cpp **-U**__LINE__ -o main
- It is possible to redefine a macro. First undefine it and then define it using #define.

Preprocessor Directives - Conditional Compilation

- **#if**, **#ifdef**, **#ifndef**, **#elif** and **#else** are used to compile code conditionally.
  - A conditional preprocessing block starts with **#if**, **#ifdef** or **#ifndef** directive, then optionally any number of **#elif** directives, then optionally at most one **#else** directive and is terminated with **#endif** directive. In a condition logical operators can be used.
  - It is possible to have nested blocks. Any inner blocks are processed separately.
  - To comment code out we shall put the code between **#if 0** and **#endif**
- Multiple inclusion guard: prevent multiple inclusion of a header file.
  - We can use a macro (e.g. MYLIB_H) and check if it has not been defined, we define it using #define. #ifndef shall be ended with #endif. In this way the block between #ifndef and #endif will be included in the actual code only once

```
// Inclusion guard
#ifndef MYLIB_H
#define MYLIB_H

int func(void) { return 100; }

#endif /* MYLIB_H */
```

Preprocessor Directives - Operators

- The stringify operator (**#**) is used to convert a macro argument to a string.

```cpp
#include <iostream>
#define print(exp) std::cout << #exp << " = " << (exp) << std::endl;
int main(void) { print(10 * 20); return 0; }
```

  - Output: 10 * 20 = 200

- The concatenation operator (**##**) joins its left and right operands into a single token.

  Output: **Hello World!**

```cpp
#include <iostream>
#define TEXT_A "Hello "
#define TEXT_B "World!\n"
#define print(X) std::cout << TEXT_##X;
int main(void) { print(A); print(B); return 0; }
```

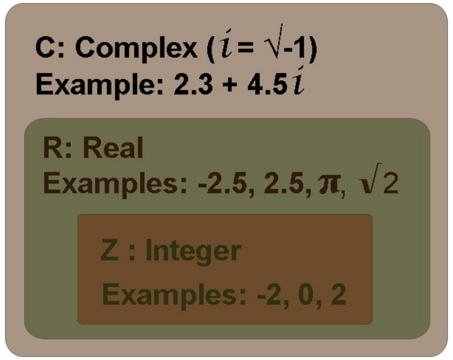
- The **defined** Operator can be used in the conditions of **#if** and **#elif**
  - **#ifdef identifier** is equivalent to **#if defined(identifier)**
  - **#ifndef identifier** is equivalent to **#if !defined(identifier)**

```cpp
#if defined(__linux__) && defined(__GNUC__)
    std::cout << "GNU/Linux" << std::endl;
#endif
```

- **#error [message]** can be used to generate an error and stop compilation.

C++ Types

- An object refers to a location in memory whose content represent data or code. 🔗 [Types]
- An object type specifies the amount of occupied memory and how its data is stored in
  - E.g. in four bytes it is possible to store an integer, a float, 4 characters strings and ect.
- There are different types in C++
  - Fundamental types: **boolean**, **integer**, **floating point** and **void**
    - **void:** an incomplete type
    - Boolean (true or false)
    - Integer types: signed and unsigned
    - Floating point types(defined in IEEE 754 standard, if supported)
      - Single precision and double precision
  - Compound types: **enumeration**, **array**, **union**, **structure**, **pointer**, **reference**, **function**, **function pointer** and **class**.



Different types of numbers

Fundamental Types

Data type	Keyword	Size	Range
Character	char	At least 1 byte	-128 to 127
Integer	int	At least 2 bytes	For 4 bytes: -2,147,483,648 to 2,147,483,647
Single precision floating-point	float	4 bytes	$\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$
Double precision floating point	double	8 bytes	$\pm 2.23 \times 10^{-308}$ to $\pm 1.8 \times 10^{308}$
Boolean	bool	At least 1 byte	false or true
Valueless and typeless	void	An incomplete type	

Wide character types (char16_t, char32_t, wchar_t) are integer types. Size of wchar_t is at least 2 bytes

- Type modifiers (signed, unsigned, short and long)
  - To change size and signedness of the fundamental types
  - By default the char, int, float and double data types are signed (can cover ± numbers)
    - The char data type is used for plain characters. E.g. 'A'
    - Use signed only with char when data is not considered as a character

Fundamental Types

- unsigned can only be used with int and char data types
- long can only be used with int and double data types
- short can only be used with int data type
- Type modifiers shall be written before the data types. E.g. unsigned short int var = 0;

Data type	Size	Range
(signed) char	At least 1 byte	-128 to 127
unsigned char	At least 1 byte	0 to 255
short int	At least 2 bytes	For 2 bytes: -32,768 to 32,767
unsigned short int	At least 2 bytes	For 2 bytes: 0 to 65,535
unsigned int	At least 2 bytes	For 4 bytes: 0 to 4294967295
long int	At least 4 bytes	For 4 bytes: -2,147,483,648 to 2,147,483,647
unsigned long int	At least 4 bytes	For 4 bytes: 0 to 4294967295
long long int	At least 8 bytes	-9,223,372,036, 854,775,808 to 9,223,372,036, 854,775,807
unsigned long long int	At least 8 bytes	0 to 18,446,744,073, 709,551,615
long double	16 bytes	$\pm 3.36 \times 10^{-4932}$ to $\pm 1.18 \times 10^{4932}$

Fundamental Types

- The standard specifies only minimum sizes for the fundamental types.
  - The sizeof operator can be used to get the actual size.
    - sizeof(*type*) or sizeof expression. E.g. sizeof(long int)
    - sizeof is a compile time operator and it is evaluated during compilation
  - You can use the **limits** header file to get min and max of the fundamental types.
  - Exact width integer types have been defined in the **cstdint** header file.
  - int8_t, int16_t, int32_t and int64_t: signed integer types with exact 8, 16, 32 and 64 bits
  - uint8_t, uint16_t, uint32_t and uint64_t: unsigned integer types with exact 8, 16, 32 and 64 bits
- Impossible to create objects of type void.
  - void is used when a function has no parameter or returns nothing. E.g. void func(void);
  - A void pointer(void *) is used to point to an object regardless of its type.
  - To discard an expression we can convert it to void. E.g. (void)printf(“Hello World!”);

Declaration

- A declaration specifies the properties of one or more identifiers
- Identifiers you declare can be names of variables, functions, and etc. E.g. int var;
- You know that identifiers have their own scope
- Generally identifiers shall be declared before we use them.
  - Exceptions: Labels, macros, tags and enumeration constants.
- There are different kinds of declaration. For example:
  - Declaration of labels, macros, tags and enumeration constants.
  - Declarations of one or more variables or functions
  - typedef and using declaration; which declare an alias for an already existing type
    - typedef type identifier; E.g. typedef unsigned char data_t;
    - using identifier = type; E.g. using data_t = unsigned char;

Storage Classes

- The general form of variable declaration: **type variable [, variable [, ...]]**; E.g. int a, b;
- Generally there are three types of scope:
  - Local: variables whose scope is limited to a language construct.
  - File(global but static): variables declared out of all language construct in a file
  - Program (global but not static): variables which can be accessible in different files
- A **storage class** determines where a variable is stored and the scope of the variable
  - There are five storage classes: **auto, extern, register, static** and **mutable**
  - If we use a storage class, it shall be written before the type. E.g. static char chr;
- The default storage class of local variables is **auto**. Local variables shall be initialized!
- The **register** storage class with a local variable hints to the compiler to place the object in a processor's register. It has been deprecated in C++17. No need to use it.

Storage Classes

- The static storage class with a variable makes the variable like a global variable but with scope of file or function. Linkage of static objects is internal.
- The extern storage class is used to declare a reference to a **global** object defined in another file. Linkage of extern objects is external.
  - We can not use extern to get access to a file scope variable whose storage class is static.
  - extern "C" is used to specify C linkage for objects with external linkage.
    - E.g. `extern "C" void func(void);`
- mutable permits modification of a class/struct/union member declared mutable even if there is a const instance object.
- A type qualifier is used to modify properties of an object.
  - C++ type qualifiers: const and volatile

Type Qualifiers

- The const type qualifier is used to make runtime constant objects.
  - Such objects must be initialized and can not be changed after initialization
    - Possible to change! But we shall not try to change.
  - E.g. int var; std::cin >> var; const int cvar{var};
  - constexpr is used to make compile-time constants. E.g. constexpr double PI{3.1415};
- The volatile type qualifier is used to make variables read always from memory.
  - Unlike register storage class. The compiler does not optimize such variables
  - The volatile type qualifier is used to declare shared variables
    - Which can be modified by multiple routines, e.g. an interrupt
  - E.g. const int var = a; volatile int status; const volatile int var = a; and etc.
- To get the address of a variable in the memory the & operator is used.
  - E.g. int var = 10; std::cout << &var << std::endl;

Literals

- A literal is a hard coded constant. E.g. 123, "Hello World!" and etc.
- A literal can be a an integer, a floating-point number, a character, a string or etc.
- Type of a literal is determined by its value and its suffix.
- Integer literals
  - Type of an integer literal by default is int. E.g. 2021
  - An unsigned int literal can be expressed using the suffix u or U. E.g 2021U
  - A long int literal can be expressed using the suffix l or L. E.g 2021L
  - An unsigned long int literal can be expressed using the suffix ul or UL. E.g 2021UL
  - A long long int literal can be expressed using the suffix ll or LL. E.g 2021LL
  - An unsigned long long int literal can be expressed using the suffix ull or ULL. E.g 2021ULL
  - Single quotes (') may be inserted between the digits as a separator. E.g. 20'000'000

Literals

- A character literal is a character enclosed by single quotation marks. E.g. 'A', '\n', '\\'
- A String literal is a sequence of characters enclosed by double quotation marks.
  - E.g. "Hello World!"
- Floating-Point Constants
  - The default type of a real number is double. E.g. 3.1425
  - A float literal can be expressed using the suffix f or F. E.g 3.1425F
  - A long double literal can be expressed using the suffix l or L. E.g 3.1425L
  - Single quotes (') may be inserted between the digits as a separator. E.g. 3.14'15'92
- The prefixes 0b/0B, 0 and 0x/0X are used to express numbers in binary, octal and hex forms
  - 0b10101010U is a binary unsigned int
  - 0x2fUL is a hexadecimal unsigned long int

Variable Initialization

- Variables can be initialized in different ways
  - Copy initialization using the = operator. E.g. `int a = 3;`
    - Copies value of the right-hand side to the variable on the left-hand side
  - Direct initialization using parentheses. E.g. `int width( 5 );`
  - List/uniform initialization using curly braces is done in two ways
    - Direct list initialization. E.g. `int width { 10 };`
    - Copy list initialization. E.g. `int width = { 10 };`
    - In list initialization narrowing conversion is not allowed. E.g. `int width{4.5}; // error`
    - Zero initialization can be done using {} or {0}. E.g. `int a{}, b {0};`
  - Best practice: use uniform initialization whenever possible.

Operators & Expression

- An expression consists of a sequence of constants, identifiers, function calls and operators
  - Which is evaluated by performing the operations. E.g. **(12 + (x * y) / z**
- Every expression has a type and it is the type of the evaluated value of the expression.
  - If an expression has no value, its type is void.
  - E.g. int a = 3; float b = 2.5f; char ch = ‘a’; **(ch + a * b) / (b + a)** => its type is float
- An expression containing more than one operator is evaluated according to operators
  - **Precedence** which specifies priority order of operators
    - E.g. a + b * c is evaluated as a + (b × c) not (a + b) × c
  - **Associativity** which specifies evaluation direction of operators
    - In an expression that contains multiple operators with the same precedence, associativity specifies an operand is grouped with the one on its left or the one on its right.
    - E.g. a * b / c; is evaluated as (a × b) / c and a = b = c; is is evaluated as a = (b = c)

🔗 [Operator Precedence]

Operators & Expression

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

Operators & Expression

- Division of two integers by default is an integer division.
- If divisor of a division is zero the behavior is undefined. If the division is floating point:
  - If **cmath** has implemented **INFINITY**, the result is **INFINITY**.
  - In **cmath** NAN and INFINITY may be implemented.
- In x % y the operands shall be integers and if y is zero, the behavior is undefined.
- Operands of bitwise operators shall be unsigned integers.
- A compound assignment of x operator= y is equivalent to x = x operator (y)
- If Postfix inc/dec(x++ and x--) x is used in an expression, it is changed after it is used.
- If Prefix inc/dec (++x and --x) x is used in an expression, it is changed then used.
- The evaluation order of operands of most operators is undefined.
  - The &&, ||, ?: and comma(,) operators have defined evaluation orders

Type Conversion

- Type casting means converting a data type into another one (type conversion)
- Type conversion is required when operands of an operator are of different types
  - E.g. char a = 'A'; float b = 12.0f; double c = a + b; // Implicit type casting
- In C++ there are two types of type casting
  - **Implicit** type casting which is done automatically by the compiler
    - Is also called as standard type conversion
  - **Explicit** type casting which is done using operators
- Implicit type casting when types mismatch
  - Conversion without changing the significance of the values stored inside the variable
  - The compiler promotes lower types to higher compatible types according to
    - bool, signed/unsigned char and short int ➤ int (done automatically)
    - int ➤ unsigned int ➤ long int ➤ unsigned long int ➤ long long int ➤ unsigned long long int (done if needed)
    - unsigned long long int ➤ float ➤ double ➤ long double (done if needed)

Type Conversion

- Implicit type casting also occurs also in
  - Assignments and initializations. The right side operand is converted to the type of the left side operand
  - Function calls. The passed arguments are converted to the types of the corresponding parameters
  - In return statements. The value of a return expression is converted to the function's return type.
  - In an expression used in an **if** statement or a **loop** (bool).
  - In an expression used in a **switch** statement (an integral type).
- Implicit type casting of a higher type to a lower type can cause problems
  - May lose values. E.g. `int x = 10.5;`
  - May lose signedness. E.g. `unsigned int x = -5;`
- g++ **-Wconversion** ... reports all the warnings related to implicit conversion
- ***Generally we shall avoid implicit type conversion. We shall use explicit type casting***

Explicit Type Casting

- **Explicit** type casting is used to force type conversion
- C++ supports different types of explicit type casting
  - C-style cast. E.g. int x{10}, y{4}; double d{(double)x / y};
    - ***Don't use the C-style type casting operator***
  - Static type casting using the static_cast operator.
    - E.g. int x{10}, y{4}; double d{static_cast<double>(x) / y}; // To have floating point division
  - const_cast: can be used to remove const and volatile qualifiers of any pointer or reference
  - reinterpret_cast: is used to convert a pointer type to another pointer type.
    - E.g. double average{3.5}; std::uint8_t *ptr{reinterpret_cast<std::uint8_t *>(&average)};
  - dynamic_cast: is used to convert pointers and references to classes up, down, and sideways along the inheritance hierarchy during runtime.

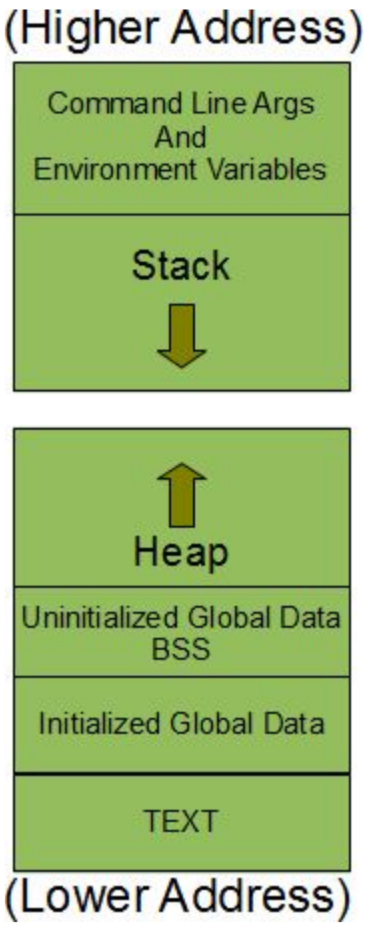
Memory Layout and Variable Life Time

- Memory Map of a C++ Program
  - **Code segment (text)**: Contains
    - The program code (read-only and shareable)
  - **Data segment**: Contains
    - Initialized global and static variables (**DS**)
      - Read only segment. E.g. const and string literals
      - Read-write segment. E.g. static int var = 10;
    - Uninitialized global and static variables (BSS)
      - They get initialized by the compiler to zero
  - **Heap**: Used for dynamic memory allocation/deallocation
  - **Stack**: Used for
    - Local and temporary variables (auto)
    - Function arguments and calls

Memory Layout

BSS (Block Started by Symbol) includes the uninitialized global and static variables. **DS** includes the initialized constants global and static variables and strings.

Run ***size app*** command to get the memory info of **app**



Compound Types - Enumeration

- An enumeration is an integer type and is defined using the enum keyword.
  - Makes a type for constants of the same type. E.g. enum Color { RED = 1, GREEN, BLUE };

  enum struct|class identifier : type { enumerator = constexpr , enumerator = constexpr , ... };
- The **identifier** is a tag name and it can be omitted. Tags have their own namespace.
- The enumerators are compile time enumeration constants and their default type is int.
  - Possible to change its type to any integer type. E.g. enum Color : std::uint8_t { … }
- Values of the constants start with 0 and is incremented automatically by one.
- It is possible to change the default values of the constants.
  - E.g. enum Color {RED, GREEN = 5, BLUE}; // RED is 0, BLUE is 5 and GREEN is 6
- Different constants in an enumeration may have the same value. But value of an implicitly-specified enumeration constant shall be unique.

Compound Types - Enumeration

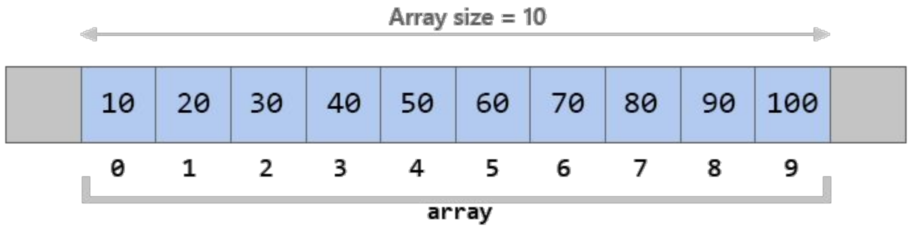
- Scope of enumerators in an enum by default starts after they appear in a code.
  - The scope resolver operator (::) also can be used to get access to an enumerator
  - E.g. Color color{GREEN}; color = Color::BLUE;
- To create scoped enumerations you can use class or struct. Generally scope of an enumeration or an enumerator starts after it appears in its parent scope.
  - Look at **UNKNOWN** in **Color** and **Status**.

  To make enumerations both strongly typed and strongly scoped, scoped enumerations can be used.

```cpp
#include <iostream>

enum struct Color : std::uint8_t { RED, GREEN = 6, BLUE, UNKNOWN};

int main(void) {
  enum Status { ERROR, WARNING, OKAY, UNKNOWN = OKAY + 100};

  Status status{OKAY};
  status = Status::WARNING;
  std::cout << status << std::endl;

  Color color{Color::GREEN};
  std::cout << static_cast<int>(Color::BLUE) << std::endl;
  std::cout << ((color == Color::BLUE) ? "Blue" : "Green") << std::endl;

  return 0;
}
```

Compound Types - Array

- An array is the simplest data structure which contains elements, stored sequentially in a continuous piece of memory.
- Elements in an array have the same
- data type and it can be any object type



int array[10] {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};

- An array can be defined by an identifier, type of the elements, and number of the elements in the array; i.e. **type name[number_of_elements];**
  - Number of elements shall be an integer greater than zero. It shall a compile time constant.
- To get size (occupied memory) of an array, use the sizeof operator. E.g. *sizeof(array);*
- To get number of the elements in an array you can divide
  - Size of the array by size of the array type or an element; i.e. *sizeof(array) / sizeof(array[0])*

Compound Types - Array

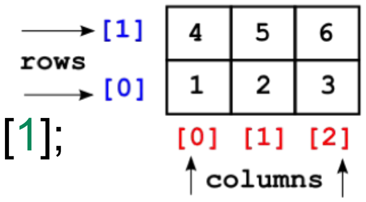
- Using an **index** and the subscript (**[ ]**) operator we can get access to an element
  - The index is an integer greater or equal to zero
  - Indexes are in the range **zero** and **length of the array - 1**
    - Index of the **first** element in an array is **zero**. E.g. array[0]
    - Index of the **last** element in an array is **length - 1**. E.g. array[9]
  - Name of an array **addresses** the first element; i.e. **&array[0]. it is like a const pointer**
- C++ does not provide boundary checking and you need to ensure that an index is not out of the boundaries. E.g. int array[4]{10, 20, 30, 40}; array[10] = 100; // not OK
- You know the general rule of initialization
  - Uninitialized global and static variables are initialized by the compiler to **zero**
  - You have to initialize local variables

Compound Types - Array

- To initialize all the elements of an array use zero initialization.
  - E.g. int array[5]{}; or int array[5]{0};
- As the best practice always state length of arrays when you initialize it.
  - int arr1[**5**] = {1, 2, 3, 4, 5}; int arr2[] = {1, 2, 3, 4, 5}; // Length of the arr1 and arr2 is 5
  - If number of initializers is greater than length, an error is generated
- If an array is partially initialized, the **uninitialized** elements are automatically set to **zero**
- It is possible to use typedef or using to create an array type.
  - E.g. typedef int array_t[3]; or using array_t = int[3]; then array_t array{1, 2, 4};
- Arrays can be multidimensional(array of arrays), just add another subscript operator

E.g. int matrix[2][3]; or array_t matrix[2]; // A 2D array which has 2 rows and 3 columns

Initialization: int matrix[2][3]{}; int matrix[2][3]{{1, 2, 3}, {4, 5, 6}}; std::cout << matrix[0][1];



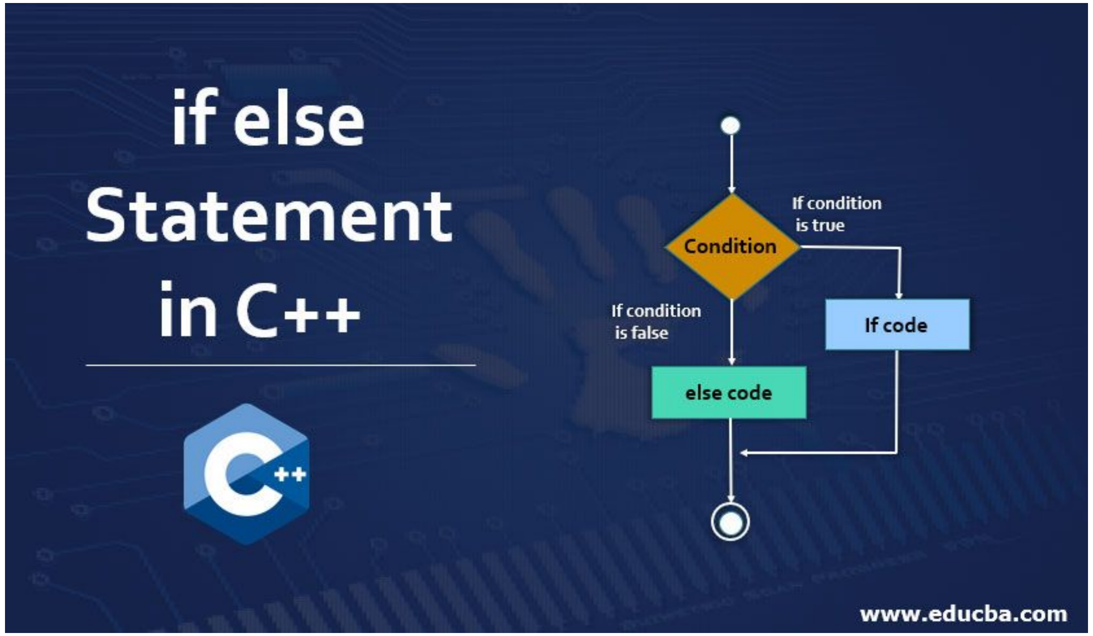
Compound Types - Arrays and Strings

- The standard library provides std::array in the **<array>** header to make safer arrays.
  - If we use the **at()** function with an invalid index an exception is thrown. For example:

    `std::array<int, 5> array{1, 2, 3, 4, 5}; std::cout << array.at(20) << std::endl; // Throws an exception`
- A string is a null(**'\0'**) terminated sequence of characters stored in an **array** of type **char**
- **Length** of a string is the number of characters before the null(**'\0'**) character
  - E.g. `char str[6]{'H', 'e', 'l', 'l', 'o', '\0'}; // Length of the string is 5`
- A string can be initialized using a string literal. E.g. `char str[10]{"Hello"};`
- For string handling **cstdio**, **cstring**, **string** and **sstream** are provided by the standard library
  - **E.g.** *sprintf, sscanf* and etc. in **cstdio**. And *strlen, strcat, strcpy* and etc. in **cstring**
  - std::string in **string** is the C++ template class to store and manipulate strings
  - std::stringstream in **sstream** has implemented input and output operations on string

Statements

- A statement is a command which makes a computer to take one or more specific actions
  - E.g. assigning a value to a variable, calling functions, jumping to another statement and etc.
- A computer program is made up of a series of statements.
- Statements in C++
  - Declaration statements
  - Labeled statements
  - Expression Statements
  - Compound Statements
  - Selection Statements
  - Iteration Statements
  - Jump Statements
  - Try Blocks



Statements

- A **declaration statement** introduces, creates, and optionally initializes one or several identifiers, typically variables. E.g. int var{30};
- A label is an identifier followed by a colon (**:**). E.g. **exit:**
- A **labeled statement** is preceded by a label.
  - A simple label can be target of a goto statement. E.g. goto exit;
  - case labeled statements in a switch statements
  - default labeled statements in a switch statements
- A **compound statement** consists of multiple statements and declarations within curly brackets ({}). I.e. *{ [list of declarations and statements] }*
  - Usually the declarations are placed at the beginning
  - E.g. body of the **main** function in a program
  - It is also called as block statement

```
switch (expression)
{
case CASE_1:
    /* Statements */
    break;

case CASE_2:
    /* Statements */
    break;

default:
    /* The default statements */
    break;
}
```

Statements

- An **expression statement** consists of an optional expression followed by a semicolon

    [expression] ; For examples: a = 2 * b + c; printf(“Hello World”); and etc.

    - If no expression exists, the statement is often called a **null** statement.
- **Selection Statements**
    - Are used to direct the flow of execution along different branches depending on a condition
    - Are also called **branching** and there are two types of selection statement; ***if*** and ***switch***
- An **if** statement is in the form of ***if (expression) statement_1 [else statement_2]***
    - The expression shall be a **boolean** and varying. The expression is evaluated first
        - If it is evaluated to true then **statement_1** is executed**;** otherwise **statement_2** (if exists)
    - The statements shall be **compound statements**; i.e. enclosed in **{}**
    - The **else** clause is optional and it can be **immediately** followed by an **if** statement

Statements

- It is possible to have nested and cascaded if statements
  - if … else if constructs shall be terminated with an else statement
- A **switch statement** compares value of the expression with multiple cases. Once a case match is found, the statement associated with the case is executed. If there is no match, the default statement will be executed.
  - The expression must be of integral or enumeration type
  - Every switch statement shall have a default label
    - The default label shall appear as the first or the last label
  - The expression shall not have essentially **boolean** type
  - It is possible to have nested switch statements

```
switch (expression)
{
case CASE_1:
    /* Statement */
    break;

case CASE_2:
    /* Statement */
    break;

default:
    /* Statement */
    break;
}
```

```
if (expression)
{
    if (expression)
    {
    }
    else
    {
    }
}
else if (expression)
{
    if (expression)
    {
    }
}
else
{
}
```

Statements

- The case labels shall be integral or enumeration constants; i.e. ***case* constant:** *statement*
- A switch statement shall have at least two switch-clause statements
- An unconditional break statement shall terminate every switch-clause statement
- In a multiple choice situation where only one branch is chosen, instead of multiple if..else statements it is better to use a switch statement
  - Your code will be more readable and maintainable
  - Generally a switch-statement has a better performance than multiple if..else statements
- Note that a break ends a switch-clause statements
- If you need to declare and initialize a variable in a switch-clause statement, use a compound statement.

```
// This is NOT OK
switch (expression)
{
default:
    /* Statement */
    break;
}

// This is NOT OK
switch (expression)
{
case CASE_1:
default:
    /* Statement */
    break;
}
```

```
// This is OK
switch (expression)
{
case CASE_1:
case CASE_2:
    /* Statement */
    break;

default:
    /* Statement */
    break;
}
```

Statements

- An **iteration statement** is used to execute a group of statements repeatedly in its body
  - Body of an iteration-statement shall be a compound-statement; i.e. enclosed in **{}**
- There are four kinds of loop; while, do… while, and for and *range-based* for
  - The number of iterations is controlled by a **condition** which is called **controlling expression**
    - The controlling expression shall have essentially **boolean** type
    - The body repeatedly is executed as long as the controlling expression is true
  - It is possible to jump out or back to the top of loops using
    - Jump statements; i.e. break, continue, return and goto
    - We shall not have more than one break or goto statement to terminate an iteration
  - It is possible to have nested iteration statements
  - A range-based for loop has no condition.

Statements

- In a while statement first the controlling expression is evaluated.
  - If its value is true then the body will be executed
  - If the expression is always **true**, we have a forever loop

```
while (1)
{
    /* body */
}
```

```
while (expression)
{
    /* body */
}
```

- A do…while statement executes the body statement once before evaluation of the controlling expression (at least one iteration of the body is performed)

```
do
{
    /* body */
} while (expression);
```

- In a for statement there are three optional expressions
  - Generally for loops are used when we have a counter to control the loop

```
for ([initialization expression]; [controlling expression]; [update expression])
{
    /* the body of the for statement */
}
```

Statements

- The expressions in the head of a for loop are optional.
  - If none of them exists, we have a forever loop

```
for (;;)
{
    /* body */
}
```

- The **initialization expression** is used to perform any necessary initialization.
  - It is evaluated only once, before the first evaluation of the **controlling expression**.
  - Shall assign a value to the loop counter, or define and initialize the loop counter
  - Multiple loop variables can be declared and initialized using comma operator
- The **controlling expression** is tested before each iteration.
  - Loop execution ends when this expression evaluates to false.
  - Shall use the loop counter and optionally boolean loop control flags
- The **update expression** is performed after each iteration and before the controlling expression is tested again. It is used to update the loop counters (incrementation and decrementation)

Statements

- Variables declared in the **initialization expression** of a for loop are local
- In the **update expression** it is possible to use the comma operator to update multiple variables.

```
for (int i = 0, j = 100; i < 10 && j > 0; i++, j--)
{
    /* body */
}
```

  In this expression we shall not use objects that are modified in the for loop body
- Type of a for-loop counter shall not be a floating point type
- A for-loop counter shall not be modified in the loop body
- A **for-each** loop is used to iterate over a range of values
  - E.g. all elements in a container like an array.

    for(range-declaration : range-expression) { }

```
int array[5]{1, 2, 3, 4, 5};
for (const int elem : array) {
    std::cout << elem << '\t';
}
std::cout << std::endl;
```

Statements

- Jump statements: break; continue; goto label; return [expression];
- A break statement is used to jump to the first statement after a loop or a switch.
- A continue statement can only be used in the body of a loop to jump to the head of the loop
- A goto statement is used to jump unconditionally to a labeled statement in the same function
- ***We shall avoid using goto statements!***
- A return statement ends a function and jumps back to where the function was called
  - If there is an expression, its value will be returned to the caller
- A function should have no more than one return statement

```cpp
for (int i = 0; i < 10; i++) {
  std::cout << i << '\t';
  if (i == 5) { break; }
}
```

```cpp
for (int i = 0; i < 10; i++) {
   if (i % 2 == 0) { continue; }
   std::cout << i << '\t';
}
```

Statements

- Condition of a while or an if statement can be declaration of a single non-array variable with a brace-or-equal initializer. ***But we shall not use this feature!***
  - Scope of such a variable is limited to the while or if statement.
  - In a while loop the expression is evaluated before each iteration

```
while (int a = func()) { /* statements */ }     if (int a = func()) { /* statements */ }
```

- Since C++ 17 it is possible to have an init-statement in **if** and **switch** statements.
- In the init-statement we can declare and initialize variables whose scope is limited to the if or switch statements or it can be an expression statement.
- Possible to make constexpr if statements.
  - In a constexpr if statement the condition shall be a compile-time constant expression.

```
if constexpr (sizeof(int) == 4) { std::cout << "constexpr" << std::endl; }
```

```
switch (int a{0}; exp) { … }
int a{0}; switch (a = 10; exp) { … }
```

```
if (int a{0}; exp) { … }
int a{0}; if (a = 10; exp) { … }
```

Compound Types - Structures

- Structures are a way to aggregate data together in order to
  - Define new data types based on existing data types
  - Make abstract types. E.g. date, person etc.
- A structure is defined using the struct keyword

  struct [identifier] { member_declaration_list };

  - The identifier is a tag name and is optional
    - Tag names have a different namespace from variables and functions. The compiler can distinguish tag names from the other identifiers. Therefore it is possible to have for example a variable with the same tag name of a struct.
  - The variable members are declared just like local variables
    - A struct makes a scope and its possible to have the same identifier in two different structs.
    - E..g. std::cout << sizeof(person::age) << std::endl;

```
struct date {
    int day;
    int month;
    int year;
};
struct person {
    int id;
    int age;
    char name[32];
    date birthdate;
};
```

Compound Types - Structures

- A variable member of a struct cannot be of type of the struct itself.
- An instance of a struct type can be declared: ***[struct]* struct_name *variable_name;***
- An instance of a struct can be initialized in different ways
  - Zero initialization: date today{}; or date today{0};
  - Partial initialization: date today{18, 1}; The uninitialized members will be set to zero
  - Full initialization: date today{18, 10, 2023};
    - As the best practice; initialize arrays and structures using zero or full initialization.
  - *Members are initialized in order*
  - Initialize a new instance using an already existing instance: E.g. date d{today};
- To get access to the members of a struct object the **dot** operator is used.
  - E.g. date today{}; today.day = 18; std::cout << today.day << std::endl;

Compound Types - Structures

- It is possible to copy a struct object to another one. They shall have the same type.
  - E.g. date today{18, 1, 2023}, d; d = today; std::cout << d.day << std::endl; // 18
- To get size of a struct or a struct member we can use sizeof and **::** operators
  - std::cout << sizeof(date) << ", " << sizeof(date::day) << std::endl;
- It is possible to use typedef and using to make an alias for a struct type.
  - E.g. using date_t = date; or typedef struct date date_t;
- Possible to have nested and unnamed structs.
- A struct object can have any storage class and type qualifier. E.g. static const date today;
  - Members of a const struct object is immutable. To make a member mutable use mutable.

    E.g. struct A { mutable int m; int n; }; const A a{1, 2}; a.m = 1; // m is changed
- A struct member can be static, mutable, and have any type qualifier.

Compound Types - Structures

- Members of a struct can be initialized using default initializers in the struct
  - **Non-const static** members shall be initialized out of the struct and in the global scope.

```cpp
#include <iostream>

struct Data {
  static constexpr int a{100};
  static const int b{5}; static int c;
  mutable int d{10}; const volatile int e{20};
  const int f{30}; int g{60};
};

int Data::c{0}; /* Initialize the static member */

int main(void) {
  Data data{1, 2, 8};
  data.d = 100; /* mutable */
  std::cout << data.a << ", " << data.b << ", " << data.c << ", "
            << data.d << ", " << data.e << ", " << data.f << std::endl;
  return 0;
}
```

- **Bit-fields** are special type of structs to store data in a compact way.
  - Member declaration: **type bitfield: width;**
  - **type:** can only be an integer type.
  - **width**: number of bits occupied by the bit-field
  - Impossible to get **sizeof** and **address** of a bit-field

```cpp
struct Date { uint32_t day : 5; uint32_t month : 4; uint32_t year : 23; };

Date today{18, 1, 2023};
std::cout << today.year << "-" << today.month << "-" << today.day;
```

Compound Types - Union

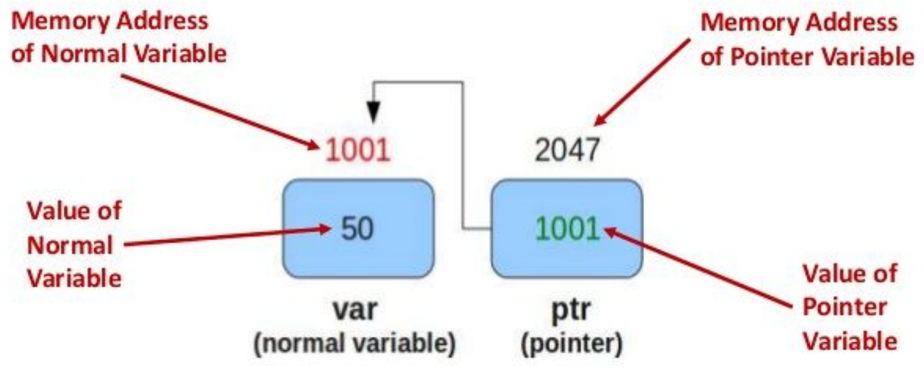
- A union is a special type that can hold only one of its **non-static** data members at a time.
  - All members share the same memory location and we can get access the same data in different ways. A union is at least as big as necessary to hold its largest data member
- A union can be defined using union keyword; E.g. *union [identifier] { member_declaration_list };*
- A union is similar to a struct. ***But when we initialize it we can have only one initializer.***
- It is possible to assign a union variable to another, but with the same type
- A union member can be static or mutable, or have any type qualifier.
- Like a struct it is possible to initialize a union using a default initializer
  - But only one non-static member can be initialized

```
#include <iostream>
union Data { uint32_t dword{0}; uint16_t word[2]; uint8_t byte[4]; };
int main(void) { Data data{0x12345678}; std::cout << std::hex << data.dword << ", " << data.word[0] << ", " << data.word[1] << ", "
                                            << data.byte[2] << ", " << static_cast<int>(data.byte[3]) << std::endl; return 0; }
```

Compound Types - Pointer

- A pointer is like a reference to an object.
  - A pointer to a variable can be declared using the * operator as

    type *pointer_name [= initializer];
  - E.g. int var = 50; int ***iptr** = &var; // iptr points to var
- A pointer basically ***holds an address*** and even ***has an address*** like a variable
  - E.g. `std::cout << "Address of var: " << &var << "\nValue of iptr: " << iptr << "\nAddress of iptr: " << &iptr;`
  - So sizeof all pointer types is the same and depends on the architecture of the system
- Pointers are generally used to get indirectly access to variables and functions in order to improve performance and memory usage



Compound Types - Pointer

- Pointers can be dereferenced using the * operator
  - To get access to the variables which are the target of pointers.
  - E.g. `int var = 50; int *iptr = &var; *iptr = 10; std::cout << var << " = " << *iptr; // Now the value of var is 10`
- Pointers are the most powerful feature of C++ and they can be used for example
  - Implementation of **call by address** functions
    - Without having pointers, variables are passed by values.
      - When we call functions, values of variables are copied to the arguments of functions and inside functions we have copies of variables and we can not change the original variables.
  - Implementation of dynamic data structures like linked lists
  - Instead of moving around big data in the memory, address of data can be moved or copied

```
#include <iostream>

static void func1(int x) { x = 5; }
static void func2(int *x) { *x = 20; }

int main(void) { int var = 2;
  func1(var); / A copy of var is passed to func1
  std::cout << "Value of var is " << var; // var is 2
  func2(&var); // Address of var is passed to func2
  std::cout << "Value of var is " << var; // var is 20
  return 0;
}
```

Compound Types - Pointer

- Pointers are the main source of hard to find bugs in C++ programs
- A null pointer points to nowhere and it cannot be dereferenced. E.g. int *iptr = nullptr; *iptr = 20;
- In C++ a null value (0) can be used to make null pointers. E.g. float* ptr { 0 };
  - The C NULL macro can be used to make null pointers; include **cstddef. #define NULL ((void *)0)**
  - In C++ the nullptr keyword is used as a null pointer literal to assign to any pointer
  - Variables defined as std::nullptr_t can only have nullptr as their values.
- A null pointer is always unequal to any valid pointer to an object
  - Functions that return a pointer type usually use nullptr to indicate a failure condition
  - E.g. if(nullptr != fgets(string, LENGTH, stdin)) { ... } // If fgets fails, it returns nullptr
- void pointers (void *) are used as general-purpose pointers to point to any object regardless of its type. ***We know that we can not declare variables of type void!***

Compound Types - Pointer

- A void pointer can be converted to any pointer type and vice versa.
  - The reinterpret_cast operator shall be used.
    - E.g. int var{10}; void *vptr{&var};
    - int *iptr = reinterpret_cast<int *>(vptr);
  - A void pointer cannot be dereferenced.
  - It can be used to declare a general function parameter and return types
    - E.g. void *memset(void *s, int c, size_t n) // declared in <cstring>

```
double d; Person p;
std::memset(&d, 0, sizeof(d));
std::memset(&p, 0, sizeof(p));
```

- An uninitialized pointer(**wild pointer**) points to a random location in memory space
- A pointer shall be initialized under the general initialization rules in C++
- We can use nullptr to initialize any pointer type. E.g. float *fptr = nullptr; int *iptr = nullptr;
- A pointer can also be initialized using a pointer to the same type.

Compound Types - Pointer

- A cast shall not be performed between object pointers of different types. Exceptions:
  - Converting any pointer type to a pointer to signed/unsigned char and vice versa.
    - To get access to data bytes regardless of type.
    - The reinterpret_cast operator shall be used.
  - Converting pointers to classes up, down, and sideways along the inheritance hierarchy.

```
double d{1.234};
uint8_t *ptr{reinterpret_cast<uint8_t *>(&d)};
for (int i{0}; i < sizeof(d); i++) {
  std::cout << std::hex << static_cast<int>(ptr[i]) << "\t";
}
```

- A pointer is an object and has an address so it is possible to have a pointer to another pointer. To declare a pointer to another pointer we shall use double asterisks (**)
  - E.g. `char c = 'A'; char *cptr = &c; char **dcptr = &cptr; **dcptr = 'B'; // dcptr is a double pointer.`
  - `printf("c = %c, *cptr = %c, **dcptr = %c, &c = %p, cptr = %p, *dcptr = %p\n", c, *cptr, **dcptr, &c, cptr, *dcptr);`
  - Possible to have more than 2 levels of pointer nesting. But we shall avoid using more than 2 levels

Compound Types - Pointer

- A pointer to a structure or a union
  - In two ways we can get access to the members
    - Using the dot(.) operator.
      - E.g. `(*ptr).name` // () are required
    - Using the arrow operator. E.g. `ptr->name`
    - They are equivalent.

```
struct Person { char name[32]; int age; };
Person stefan{"Stefan", 30}; Person *ptr{&stefan};
std::cout << ptr->name << " is " << (*ptr).age << " years old";
```

- Pointer Comparison and Arithmetic Operation
  - Addition and subtraction of an integer.
  - Subtracting one pointer from another.
  - Comparing two pointers
  - **Pointers know size of the type.**
    - *In the case of addition and subtraction they are moved according to size of the data type*

```
int array[10]{0};
int *ptr{array + 1}; // ptr points to the second element
*ptr = 1; // array[1] = 1;
ptr++; // ptr points to the 3rd element
*ptr = 2; // array[2] = 2;
ptr += 3; // ptr points to the 6rd element
*ptr = 5; // array[5] = 5;
ptr[3] = 7; // ptr[3] is equivalent to *(ptr + 3) which is array[8]
for (int *iptr{array}; (iptr - array) < 10; iptr++) {
    std::cout<<"Array["<<(iptr - array)<<"] = "<<*iptr<<std::endl;}
```

Compound Types - Pointer

- It is possible to compare two pointers using the ==, !=, <,>,<= and >= operators
- A pointer resulting from arithmetic operation shall address an element of the same object as that pointer operand. E.g. int arr[5] = {0}; int *ptr = &arr[0]; ptr--; // Not OK
- Subtraction between pointers shall only be applied to pointers that address elements of the same object. E.g. int a1[5] = {0}; int a2[8] = {0}; int *p1 = a1; int *p2 = a2; diff = p1 - p2; // Not OK
- The operators >, >=, < and <= shall not be applied to pointers that don’t point the same object. E.g. int a1[5] = {0}; int a2[8] = {0}; int *p1 = a1; int *p2 = a2; if(p1 > p2) {...} // Not OK
- The +, -, += and -= operators shall not be applied to an expression of pointer type
  - It is ok to use prefix and postfix ++ and --. E.g. int a1[5] = {0}; int *p = a1; p++; *p = 2;
  - Better to use array indexing. E.g. int a1[5] = {0}; int *p = a1; p[1] = 2;

Compound Types - Pointer

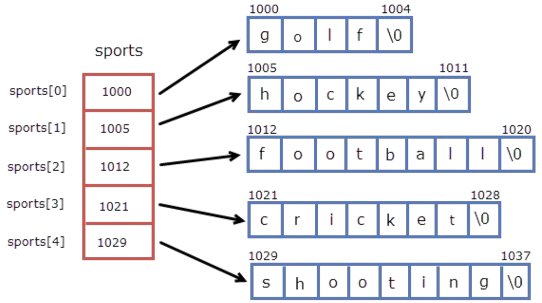
- Declaration of a pointer may contain type qualifiers. E.g. int const volatile * ptr;
- The const and volatile qualifiers may qualify
  - Either the pointer type itself, or type of the object it points to.
    - E.g. A pointer to a const variable or a const pointer to a variable.
  - If type qualifiers occur between asterisk and the pointer name, they qualify the pointer itself.
    - E.g. int var = 20; int *const ptr = &var; // ptr is a const pointer to var
    - E.g. const int var = 20; const int *ptr = &var; // ptr is a pointer to a const
    - E.g. const int var = 20; const int *const ptr = &var; // ptr is a const pointer to a const
    - E.g. int *const ptr; int * const *p2cp = &ptr; // ptr is a pointer to a const pointer
- A pointer should point to a const-qualified type whenever possible
  - The same rule shall also be applied to arrays
    - An array is a pointer

```
void func(int *ptr) // Could be void func(const int *ptr)
{
    printf("%d\n", *ptr); // The object is not changed
}
```

Compound Types - Pointer

- Name of an array is a pointer to the first element of the array.
  - E.g. int arr[5]{0}; int *ptr = arr; ptr[0] == arr[0]; const char *str = "Hello World!";
- We can create a pointer to an array as a whole (array pointer)
  - E.g. int (*ptr)[5] = &arr; // ptr is a pointer to an array of five int elements. parentheses are required
  - It is possible to define array types and array pointers.
    - E.g. using array_t = int[5]; array_t arr{0}; array_t *ptr{&arr};
  - Array pointers are useful when we deal with multidimensional arrays.
    - E.g. Name of a two multidimensional array is a double pointer.
    - E.g. int matrix[2][3]{4}; int(*ptr)[3]{matrix};
      - ***matrix** is a pointer which points to the first row.
      - **E.g. printf("maxtrix[0][0] = %d = %d = %d = %d\n",**matrix,**ptr,matrix[0][0],(*ptr)[0]);**
- We can also have an array of pointers (pointer array)
  - E.g. char *parr[5]; // parr is an array of 5 char pointers (strings)

```
array_t array{1, 2, 3, 4, 5};
array_t *ptr{&array};

for (int i{0}; i < 5; i++) {
  printf("Array[%d] = %d\n", i, (*ptr)[i]);
}
```



Compound Types - References

- In C++, there 3 kinds of variable: normal, pointer and **reference**.
- A reference basically acts like an alias of an object or value.
- A reference is declared using **&** between the reference type and the variable name
- References must be initialized and cannot be reassigned
- A reference acts like a pointer that implicitly performs indirection through it when accessed. References are internally implemented by the compiler using pointers
- C++ supports 3 kinds of reference
  - References to non-const values. E.g. `int value{5}; int &ref{value}; // reference to variable value`
  - References to const values. E.g. `const int apples{5}; const int &ref{apples}; // reference to a const`
  - r-value references. E.g. `int &&ref = 1 + 2; // An rvalue reference`
    - An r-value reference is formed by placing two ampersands after the type.

Compound Types - References

- References like pointers make no copy of the argument passed to functions
- An l-value is an expression that has an address (in memory); like variables
  - They can be on the left side of an assignment statement. E.g. int **a** = 10; a is an l-value
- An r-value is an expression on the right side of an assignment.
  - E.g. literals like **5** and **4 + 5**, and expressions like **10 * x** and etc.
- References to non-const values can only be initialized with non-const l-values
  - E.g. `int &ref3{6}; // error, 6 is an r-value`
- Reference to const value; E.g. `const int apples{5}; const int &ref{apples}; // ref is a reference to a const value`
- A reference to const can be initialized by a const and non-const l-value, and r-value.
- A reference to an r-value extends lifetime of the value. Normally r-values have expression scope; i.e the values are destroyed after using. E.g. 2 * a + 5;

Compound Types - References

- To create an r-value reference, **&&** is used. int &&ref{6 + 2};
- A reference to a struct/union/class; the dot operator is used to get access to the members.
  - E.g. struct Person { int age{}; double weight{}; }; Person person{}; Person &ref{person}; ref.age = 5;
  - An r-value reference can be converted to an l-value reference and vice versa.

```
int x{5};
const int &ref1{x}; // okay, x is a non-const l-value
const int y{7};
const int &ref2{y}; // okay, y is a const l-value
const int &ref3{6}; // okay, 6 is an r-value
```

- The compiler can deduce type of a new object from its initializer using the auto keyword.
- Type deduction drops variable type qualifiers.
  - E.g. const volatile int x{5}; auto y{x}; // type of y is int
  - To make y as a const volatile variable: const volatile auto y{x};
- Type deduction does not drop qualifiers of pointers.
  - E.g const char *str = "Hello"; auto ptr{str};

```
#include <iostream>
int main(void) {
    int x{5};   // x is a normal int
    auto &y{x}; // y is an int& reference
    auto z{y};  // z will be an "int", not an "int &"
    z = 20; y = 10;
    std::cout << x << " " << z << std::endl;
    return 0;
}
```

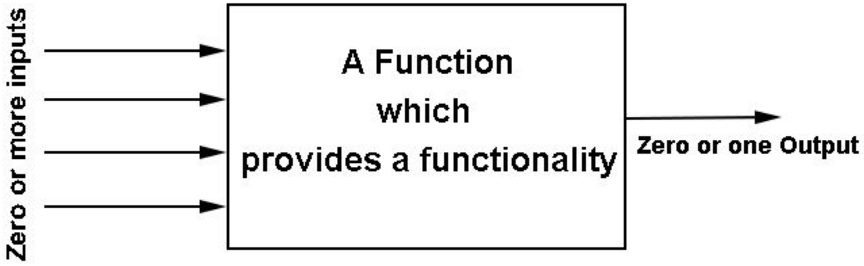
- Type deduction drops references. You can use auto& instead of auto
- Using decltype you can get type of an object or an expression. E.g. int a{0}; decltype(a) p; // p is int

Compound Types - Function

- ❖ A function is a block of organized and reusable code, used to perform a task.
- ❖ Functions provide better modularity for programs and a high degree of code reusing.
- ❖ A function is declared as

  return_type function_name(parameter_declarations);

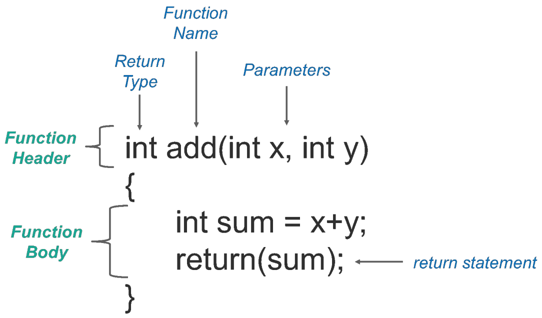
  - ➢ This is also called the function prototype
  - ➢ Functions shall be declared before using



- ❖ A function is defined (implemented) as

  return_type function_name(parameter_declarations) // The Function head

  { / Declarations and statements */ } // The function body*



- ❖ A function can be declared and defined at once.
- ❖ The ***return_type*** can be any type except **function** and **array** types
  - ➢ But we can return a pointer or a reference. E.g. char *get_input(int length);

Compound Types - Function

- The ***return_type*** can have a storage class of static or extern
  - E.g. static int add(int x, int y); or extern int add(int x, int y);
  - The default storage class of a function is extern
  - Scope of a static function is the file where it is defined in
  - You can not use extern for a function defined as static
  - Possible to deduce return types using the auto keyword
    - E.g. Such a function must be fully defined before it can be used
    - Best practice: Favor explicit return types over auto functions return type
    - Trailing return type syntax using auto and the arrow operator is possible
      - E.g. auto add(int x, int y) -> int; // Equivalent to int add(int x, int y)

```cpp
#include <iostream>

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

auto func(bool b) { return (b ?
static_cast<double>(5) : 6.7); }

auto add(int x, int y); /* Error */

int main(void) { std::cout << add(2, 3); return 0; }

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

- ***function_name*** is an identifier and follows the rules of identifiers and scopes in C++
- ***parameter_declarations*** are a comma-separated list of the parameter declarations
- If a function has no parameters, void can be used as the **parameter_declaration**

Compound Types - Function

- Parameters of a function are ordinary **local** variables. Their scope is the function block.
- A function can change the value of a parameter without affecting the value of the variable used in the function call. ***But we shall avoid modifying parameters of functions***
- We can only use register as the storage class in a function parameter declaration
- To declare an array as a function parameter:
  - As **type param_name[ ]**. E.g. void func(int len, int array[]);
  - As a pointer **type *param_name**. E.g. void func(int len, int *array);
  - As a reference **type (&param_name)[N]**. E.g. void func(int len, int (&array)[5]);
  - As a reference **type *const &param_name**. E.g. void func(int len, int *const &array);
- To declare a multidimensional array as a parameter, we shall specify size of the dimensions and only size of the first dimension can be omitted.
  - E.g. void func(int rows, int columns, int array[][**5**]);

Compound Types - Function

- Possible to have functions with variable number of arguments.
  - Such functions are called **variadic** functions. E.g. printf, scanf and etc.
  - Such functions must have at least **one mandatory** argument
  - Types of the optional arguments can also vary
- To declare a function with variable number of arguments, **...** operator is used.
  - `void func(int x, ...); // To get access to the optional arguments, macros in cstdarg can be used`
  - **We shall avoid using variadic functions**
- A function can call itself, directly or indirectly.
  - Such a function is called a recursive function.
  - E.g. `int factorial(int n) { return (n == 0) ? 1 : (n * factorial(n-1)); }`
    - Factorial of n = n! = 1 * 2 * 3 * 4 … n = n*(n - 1)!

Compound Types - Function

- We shall avoid using recursive functions
- Possible to pass arguments to the main function
  - int main(int argc, char *argv[ ]);
    - **argc** is the number of the arguments; name of the program is the first argument.
    - **argv** is an array of the arguments as strings (char *)
    - E.g. run a program like: ./program 12 hello test => you can get 12, hello and test in the main function
- A function with non-void return type shall have an explicit return statement with an expression
- A function shall not return a pointer/reference to a local object.

```
int *func(void) { int local{0}; return &local; }
```

```
int &func(void) { int local{0}; return local; }
```

- In a function declaration (prototype) it is possible to omit the parameter names
  - E.g. void func(int, int, int [ ][5]); ***But we shall specify parameter names***

Compound Types - Function

- Functions and objects used in only one translation unit shall not have external linkage.
- Declaration of a function shall use the same names and type qualifiers
  - int div(int **m**, int **n**); and int div(int **n**, int **m**) { return (n / m); } // Not OK! look at the order of n and m
- A function shall have a single exit point at the end
- A value returned by a function having non-void type or is not an overloaded operator shall be used or discarded explicitly using void type casting. E.g. (void)printf("Hello World!\n");
- It is possible to have inline functions. E.g. inline int max(int x, int y) { return (x > y) ? x : y; }
  - During compilation the machine code of an inline function is inserted where the function is called. Unlike function-like macros calls which are replaced during preprocessing.
  - Inline functions improve the performance and usually used for small blocks of code
  - The keyword inline is a request to the compiler and the compiler does not guarantee it.
    - For example recursive functions are not compiled as inline
  - inline functions are preferred to function-like macros

Compound Types - Function Pointer

- Name of a function is address of where the function starts. For example

  void func(void) {} … std::cout << reinterpret_cast<void *>(&func) << std::endl; // We can omit the & operator

- We can create function pointers. (Unlike other pointer types, a function pointer points to code, not data.)
- A function pointer is used to pass a function to another function (a callback function).
- A function pointer is declared as **return_type (*function_pointer_name)(list_of_param_types);**
  - E.g. *int (*func)(int, int);* is a function pointer which can point to any function whose return type is int and has two parameters of type int. **For example: int f(int a, int b); func = &f; // or f**
  - We can even use typedef, using or std::function in <functional> to make a function pointer type
    - int func(int x, int y){} … std::function<int(int, int)> fptr{func}; … fptr(1, 1);
    - using func_t = int (*)(int, int); **or** typedef int (*func_t)(int, int); func_t fptr{&func};
      - Then we can call func using the function pointer as **fptr(20, 30);**
  - Like normal pointers, we can have an array of function pointers. E.g. func_t farr[2] = {add, divide};

Compound Types - Function Pointer

- Return and parameter types of a function can be of function pointer type.
- Possible to have function reference. E.g. using func_t = void (&)(void);

```cpp
#include <iostream>

using func_t = void (&)(void);

void func(void) { std::cout << "func called!" << std::endl; }

void print(func_t cbptr) {
    std::cout << "Calling func ..." << std::endl;
    cbptr(); // callback function is called
}

int main(void) {
    std::cout << "Let's start ..." << std::endl;
    func_t temp = func;
    print(temp); // Calling func using a function pointer
    return 0;
}
```

```cpp
#include <iostream>
#include <functional>
typedef void (*func_t)(int);
void func(int value) { std::cout << "Value = " << value << std::endl; }
func_t get_func(void) { return func; }
void print(std::function<void(int)> fptr, int a) {
    std::cout << "Calling func ..." << std::endl;
    fptr(a); // callback function is called
}
int main(void) {
    std::cout << "Let's start ..." << std::endl;
    std::function<void(int)> temp{get_func()}; // We can omit &
    print(temp, 10); // Calling func using a function pointer
    return 0;
}
```

Compound Types - Function Overloading

❖ Function overloading in C++

- ➢ Creating multiple functions with the same name but different signatures.
- ➢ Signature of a function is the name and the params number & type; but not return type
  - ■ E.g. int add(int x, int y); ⇒ add(int,int)
- ➢ Each overloaded function has to be differentiated from the others
- ➢ Each call to an overloaded function shall be resolved to only one overloaded function.
- ➢ The compiler tries to find an exact match; if not, then it will apply a number of trivial conversions to the arguments in function calls in order to find a match; if it cannot find a match, an error is generated.

```cpp
#include <iostream>

int add(int x, int y) { return x + y; }
int add(int x, int y, int z) { return x + y + z; }
double add(double x, double y) { return x + y; }

int main(void) {
    std::cout << add(1, 2) << std::endl;     // calls add(int, int)
    std::cout << add(1, 2, 3) << std::endl;  // calls add(int, int, int)
    std::cout << add(1.2, 3.4) << std::endl; // calls add(double, double)
    return 0;
}
```

Compound Types - Default Argument Function

- C++ supports default arguments
- A default argument is a default value provided for a function parameter
- Default arguments can only be supplied for the rightmost parameters
  - E.g. void print(int x = 10, int y); // not allowed
- Default arguments can not be redeclared
- Functions with default arguments may be overloaded

```
#include <iostream>

// 4 is the default argument
void print(int x, int y = 4) {
    std::cout << "x: " << x << std::endl;
    std::cout << "y: " << y << std::endl;
}

int main(void) {
    print(1, 2); // y will use user-supplied argument 2
    print(3);    // y will use default argument 4
    return 0;
}
```

```
#include <iostream>

void print(int x, int y = 4); // forward declaration

int main(void) {
    print(1);
    print(1, 2);
    return 0;
}

// error: redefinition of default argument
void print(int x, int y = 4) {
    std::cout << "x: " << x << std::endl;
    std::cout << "y: " << y << std::endl;
}
```

Compound Types - Lambda Expression

- A lambda expression allows us to define a function inside another function.
  - The syntax is **[captureClause](parameters) -> returnType { statements; };**
- The capture clause and parameters can both be empty if they are not needed.
- The return type is optional (auto will be assumed) and parameter types can be deduced using auto
- The capture clause is used to give a lambda access to variables in the surrounding scope.
  - No need to specify them in the parameter list.

```cpp
#include <array>
#include <iostream>
#include <algorithm>

int main(void) {
    std::array<int, 6> array{1, 5, 2, 4, 8, 6};
    std::sort(array.begin(), array.end(), [](auto a, auto b) { return (a < b); });
    for (const auto elem : array) { std::cout << elem << std::endl; }
    return 0;
}
```

```cpp
#include <iostream>

int main(void) {
    // Explicitly specifying the return type
    auto divide{[](int x, int y, bool integer) -> double {
      return (integer ? x / y : static_cast<double>(x) / y); }};
    std::cout << divide(3, 2, true) << '\n';
    std::cout << divide(3, 2, false) << '\n';

    return 0;
}
```

Compound Types - Lambda Expression

- By default, variables are captured by const value
  - To modify variables were captured by value, we can mark the lambda as mutable using the mutable keyword
- Variables can also be captured by reference using **&**
  - E.g. int a = 1; [&a](int x){ a += x; }(2); std::cout << a; // "3"

```
#include <iostream>
#include <functional>
void call(std::function<void(void)> func) { func(); }
int main(void) {
    int i = 2; // i is captured by value as const.
    auto print_square = [i]() { std::cout << i * i; };
    call(print_square); // "4"
    return 0;
}
```

- It is possible to specify a default capture mode, to indicate how unspecified variables used inside the lambda, are captured.
  - A [=] means that variables are captured by value and [&] captures them by reference.
  - E.g. int a = 1, b = 1; [&, b]() mutable { b++; a += b; }(); std::cout << a << b; // "31"
- Variables may also be initialized inside the capture clause. (types will be deduced)
  - E.g. int a = 1; [&, b = 2]() { a += b; }(); std::cout << a; // "3"

Namespace

- In C++ namespaces are used to solve naming conflicts. E.g. the `std` namespace
- A namespace provides a scope (namespace scope) to the declared names within it
- Within a namespace, all names must be unique, otherwise a naming collision occurs.
- Members in a namespace are accessed using the scope resolution operator (::)
- The **using directive** tells the compiler to check a specified namespace when trying to resolve an identifier that has no namespace prefix.

```cpp
#include <iostream>

using namespace std;

int main(void)
{
   cout << "Hello world!";
   return 0;
}
```

```cpp
#include <iostream>
namespace A {
   int x{10};
}

int main(void) {
   // cout is in the std namespace and x is in A
   std::cout << A::x << "Hello world!";

   return 0;
}
```

- Best practice: Use explicit namespace prefixes to access identifiers in namespaces.

Namespace

- A **using directive** imports all of the identifiers from a namespace into the scope of the using directive
- A **using declaration** allows us to use a name with no scope as an alias for a qualified name.

```
#include <iostream>

using namespace std;

int cout() { return 5; }

int main(void)
{
    cout << "Hello, world!"; // error: reference to 'cout' is ambiguous

    return 0;
}
```

```
#include <iostream>

int main(void)
{
    // the using directive tells the compiler to
    // import all names from namespace std
    using namespace std;
    cout << "Hello world!"; // no std:: prefix is needed.
    return 0;
}
```

```
#include <iostream>

int main(void)
{
    using std::cout;  // using declaration tells the compiler that cout should resolve to std::cout
    cout << "Hello world!"; // no std:: prefix is needed! In the case of conflict, std::cout will be preferred

    return 0;
}
```

Namespace

- Problems with **using directives** and **using declarations**
- The scope of using declarations and directives statements.
  - If a using statement is used within a block, the names are applicable to just that block (block scoping)
  - If a using statement is used in the global namespace, the names are applicable to the entire rest of the file (file scope).

```
#include <iostream>

int cout() // Our own "cout" function
{
    return 0;
}

int main(void)
{
    using std::cout; // makes std::cout accessible as "cout"
    cout << "Hello, world!";

    return cout(); // error: no match for call to
}
```

```
#include <iostream>

namespace A
{
    int x{10};
}

namespace B
{
    int x{20};
}

int main(void)
{
    using namespace A;
    using namespace B;

    std::cout << x; // which x is used?

    return 0;
}
```

Namespace

- C++ supports nested namespaces and aliases. E.g. *namespace active = foo;*
- **Unnamed namespaces**: content declared in an unnamed namespace is a part of the parent namespace and has internal linkage like static objects
- **Inline namespaces** are typically used for version content

```cpp
#include <iostream>

// identical to: static void doSomething() { std::cout << "v1"; }
namespace { // unnamed namespace
    void doSomething() { // can only be accessed in this file
        std::cout << "v1";
    }
}

int main(void) {
    doSomething(); // we can call doSomething() without a namespace prefix

    return 0;
}
```

```cpp
#include <iostream>

namespace A
{
    int x{10};

    namespace B
    {
        int x{20};
    }
}

int main(void)
{
    using namespace A;
    std::cout << x << std::endl; // x is A::x

    namespace C = A::B;
    std::cout << C::x << std::endl;

    return 0;
}
```

Namespace

- Everything inside an inline namespace is considered as a part of the parent namespace
- In this example, all calls to **func** will get the **latest** version by default (the newer and better version). Users who still want the older version of **func** can explicitly call v1::func() to access the old behavior. This means existing programs that want the v1 version will need to globally replace **func** with v1::func, but this typically won't be problematic if the functions are well named.

```cpp
#include <iostream>

namespace v1 { // declare a normal namespace named v1
   void func() { std::cout << "v1"; }
}

namespace v2 { // declare a normal namespace named v2
   void func() { std::cout << "v2"; }
}
inline namespace latest { // declare an inline namespace
   void func() { std::cout << "latest"; }
}

int main(void)
{
   v1::func(); // calls the v1 version of func()
  v2::func(); // calls the v2 version of func()
   latest::func(); // calls the latest version of func()

   func(); // calls the inline version of func() (which is latest)

   return 0;
}
```

C++ Language

- Some useful links
  - C++ Reference
  - Standard C++ Library Reference
  - C++ Tutorial
  - C++ Language
  - C++ Tutorial
  - C++ Full Course For Beginners (Learn C++ in 10 hours)
  - C++ Tutorial for Beginners - Full Course
  - C++ Tutorial 2021
  - C++ Programming Tutorials Playlist