Basics

31 May 2021 15:40

Avoid complexity of all sort - make program simple and explicitly understandable

1. Initialization

Value initialization - When a variable is initialized with empty braces, value initialization . Initializes var to 0 or empty.
Direct initialization - Initialize from set of constructor args within parenthesis..
Copy initialization - initialize from another object.. Used in initializations with =, func call pass by value, func ret by val, throw/catch exception by value
Doesn't work in the case of explicit constructors if constructor args are alone provided.
List initialization/Uniform/ - use {}.. Can be x = {} or x{}. Prevents narrowing conversions
Aggregate init - Initializes an aggregate from braced-init-list. Refer struct
Reference init - initializing a reference.

https://www.learncpp.com/cpp-tutorial/variable-assignment-and-initialization/
Use explicit values for init if u r gonna use it.. Else use {} if value will be updated later.

1. Initialization stages

a. Static initialization
   i. Const - compile time
   ii. Bss init - non-local static & thread local vars
b. Dynamic
   i. Unordered dynamic init - applies to static class template static members & variable templates that aren't explicitly specialized
   ii. Parially ordered
   iii. Ordered dynamic init - applies to non-local vars. Initialization in sequence of definition within a translation unit. Initialization of static variables in different translation units is indeterminately sequenced.
c. Static local

Static local variables

Variables declared at block scope with the specifier `static` or `thread_local` (since C++11) have static or thread (since C++11) storage duration but are initialized the first time control passes through their declaration (unless their initialization is zero- or constant-initialization, which can be performed before the block is first entered). On all further calls, the declaration is skipped.

d. Local constexpr var
e. Static local constexpr var
f. Local constexpr array

arrays, more or less by definition, are addresses. So in most cases, the non-static `const(expr)` array will have to be recreated on the stack at every invocation, which defeats the point of being able to compute it at compile time.

g. h

https://en.cppreference.com/w/cpp/language/initialization

2. **Expression** -> combination of literals, variables, operators, explicit function calls -> evaluated to a single output **value**

type identifier { expression };

Expressions has two independent properties: type and value category

Primary expressions -> Operands of expressions may be other subexpressions (or) a primary expression

- Literals (e.g. `2` or `"Hello, world"`)
- Id-expressions, including
  - suitably declared unqualified identifiers (e.g. `n` or `cout`), and
  - suitably declared qualified identifiers (e.g. `std::string::npos`)
- Lambda-expressions (since C++11)
- Fold-expressions (since C++17)
- Requires-expressions (since C++20)

*** Including an expression in parenthesis makes it a primary expression. Parentheses preserve type, and value category.

Unevaluated expressions -> expressions that arent evaluated. Eg sizeof(printf("hh"))
https://en.cppreference.com/w/cpp/language/expressions

3. Endianess

"Big Endian" scheme, the most significant byte is stored first in the lowest memory address (or big byte in first addr)
"Little Endian" stores the least significant bytes in the lowest memory address (or sm all byte in first addr).

4. Types and sizes

std::nullptr_t (C++11)	Null Pointer	a null pointer	nullptr
void	Void	no type	n/a

Fundamental types and C++ guaranteed minimum size.

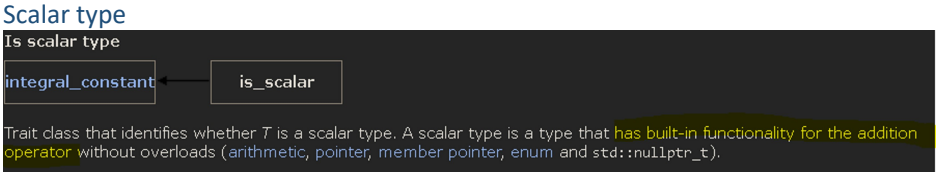
Category	Type	Minimum Size
boolean	bool	1 byte
character	char	1 byte
	wchar_t	1 byte
	char16_t	2 bytes
	char32_t	4 bytes
integer	short	2 bytes
	int	2 bytes
	long	4 bytes
	long long	8 bytes
floating point	float	4 bytes
	double	8 bytes
	long double	8 bytes

Actual sizes

```
bool:           1 bytes
char:           1 bytes
wchar_t:        2 bytes
char16_t:       2 bytes
char32_t:       4 bytes
short:          2 bytes
int:            4 bytes
long:           4 bytes
long long:      8 bytes
float:          4 bytes
double:         8 bytes
long double:    8 bytes
```

Long double - 16

5. Scalar type



6. Sizeof

`sizeof(void) = 1 byte`
warning: invalid application of 'sizeof' to a void type

7. Use nullptr instead of #include <cstddef> NULL or 0

C++11 also introduces a new type called std::nullptr_t (in header <cstddef>). std::nullptr_t can only hold one value: nullptr (use without namespace) ..
Its values are null pointer constant (see NULL), and may be implicitly converted to any pointer and pointer to member type

`sizeof(std::nullptr_t) = sizeof(void*) = sizeof(int*) = 4/8 bytes`

a. Can assign arbitrary pointers to a nullptr. The pointer becomes a null pointer and points to no data. C++ will implicitly convert nullptr to any pointer type!
b. Cannot dereference a nullptr.
c. The pointer of this type can on one hand compared with all pointers. C++ will implicitly convert nullptr to any pointer type.
d. Can be converted to all pointers of any object.
e. Can implicitly compare and convert a bool value with a nullptr. Therefore, you can use a nullptr in a logical expression.
f. **Cannot compare and convert a nullptr to an integral (not integer) type**. Bool one exception to this rule.

If 0 or NULL is used, then compiler can't tell whether we mean a null pointer or the integer 0.. For eg overloaded func(int) and func(int*) and func(NULL) is called
https://www.modernescpp.com/index.php/the-null-pointer-constant-nullptr

8. Integral type

Integral type are named so because they are stored in memory as integers, even though their behaviors might vary ..
Boolean, characters, and integer types, enumerated types)

Integer stored by a char variable are interpreted as an ASCII character.
Integer stored by a bool variable are interpreted as true/false

9. Single-quote character ' can now be used anywhere within a numeric literal for readability. Doesn't affect numeric value.

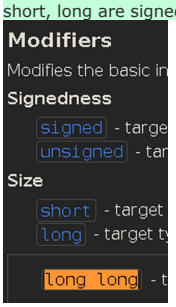
10. Signed Integers use 2's complement with sign-bit

Computers use 2's Complement Representation for Signed Integers -> Sign bit plus 2's complement of number
a. Advantage 1 -> 0 has single binary representation

b. Advantage 2 -> Addition, subtraction can be done using + operator -> addition of number and a twos complement is = to subtraction

11. Shorthand integers

short, long are signed integers..



12. Integer operations

Unary operators +x or -x will multiply by +1 or -1

'Operand type' decides operation & result type.

When doing division with two integers (called integer division), C++ always produces an integer result even if in LHS its stored in a float/double (operands matter)
If either operand is a floating point number, the result will be floating point division, not integer division (look at promotion)

13. Avoid Unsigned integers - wraparound -> 0 - 1 = 429…

Using an unsigned instead of an int to gain one more bit to represent positive integers is almost never a good idea - because of wraparounds
Ok in case of bit manipulation, embedded system, array indexing

14. Fixed width integers library cstdint

To help with cross-platform portability, C99 defined a set of **fixed-width integers** (in the stdint.h/cstdint header) that are guaranteed to have the same size on any architecture … Example std::uint8_t
Issue -> slow if not multiple of CPU address length

15. Problem uint8_t and int8_t -> char

Due to an oversight in the C++ specification, most compilers define and treat std::int8_t and std::uint8_t (and the corresponding fast and least fixed-width types) identically to types signed char and unsigned char respectively

16. std::size_t and sizeof

sizeof (and many functions that return a size or length value) return a value of type std::size_t.
std::size_t is defined as an unsigned integral type, and it is typically used to represent the size or length of objects.
std::size_t is guaranteed to be **unsigned and at least 16 bits, but on most systems will be equivalent to the address-width of the application**

17. Scientific notation std::scientific

scientific notation take the following form: *significand e exponent (or) significand* x $10^{exponent}$
Significand = one digit wide . many digits

```
std::cout << std::scientific << 602.022000000 << std::endl;
6.020220e+02
```

18. Printing numbers in different formats - hex, bin

Printing in decimal, octal, or hex is easy via use of std::dec, std::oct, and std::hex
Use bitset to print in binary

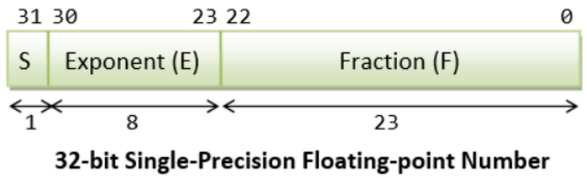
19. Floating point

Can support a variable number of digits before and after the decimal point

- floating point representation almost always follows IEEE 754 binary format. In this format, a float is 4 bytes, a double is 8
- By default, floating point literals default to type double. An f suffix is used to denote a literal of type float. Eg 5.0f
- The precision of a floating point number defines how many significant digits it can represent without information loss.
- Float values have between 6 and 9 digits of precision, **with most float values having at least 6-7 significant digits**
  23 mantissa bits => 2^(-23) => 0.0000001xxx = ~ 7 decimals after = log(2^23)/log(2) = 23
- Double values have between 15 and 18 digits of precision, **with most double values having at least 15-16 significant digits**
  52 mantissa bits => 2^(-52) => ~16 decimals after = log(2^52)
  https://stackoverflow.com/questions/13542944/how-many-significant-digits-do-floats-and-doubles-have-in-java
  Demystifying Floating Point Precision « The blog at the bottom of the sea (demofox.org)
- When outputting floating point numbers, std::cout has a default precision of 6.. Use output manipulation in <iomanip> std::setprecision(x)
- Favor double over float unless space is at a premium, as the lack of precision in a float will often lead to inaccuracies
- Avoid using operator== and operator!= with floating point operands. Use of > < >= <= are okay.

IEEE-754 Floating Point Converter (h-schmidt.net) - Sign, Exponent, Mantissa -

20. Single Precision Float



32-bit Single-Precision Floating-point Number

- S, F, E => 1 bit, 8 bit, 23 bit = 32bit
- For 1 ≤ E ≤ 254, N = (-1)^S × 1.F × 2^(E-127). These numbers are in the so-called normalized form

Eg: S=0, F = 011, E = 129

=> 1.011 = 1 + 0*2-1 + 1*2-2 + 1*2*-3 = 1.375 * 2^(129-127) = -5.5

- Fractional part (1.F) are normalized with an implicit leading 1.
- The exponent is bias (or in excess) of 127 i.e. E-127, so as to represent both positive and negative exponent. The range of E-127 is -126 to +127 for 1 ≤ E ≤ 254

- N = (-1)^S × 0.F × 2^(E-126). These numbers are in the so-called denormalized form.
- For E = 0, N = (-1)^S × 0.F × 2^(-126).
  - The exponent of 2^-126 evaluates to a very small number
  - Denormalized form is needed to represent zero (with F=0 and E=0). It can also represents very small positive and negative number close to zero.

- For E = 255, it represents special values, such as
  - ±INF (positive and negative infinity) -> F= 0, S=0 or S=1 (+ or - INF)
  - NaN (not a number) -> F = non-zero number, S= 0

21. Double Precision Float

- S, F, E => 1 bit, 11 bit, 52 bit = 64bit
- Normalized form: For 1 ≤ E ≤ 2046, N = (-1)^S × 1.F × 2^(E-1023).
- Denormalized form: For E = 0, N = (-1)^S × 0.F × 2^(-1022). These are in the denormalized form.
- For E = 2047, N represents special values, such as ±INF (infinity), NaN (not a number).

22. Bool

Size of bool is 1byte

23. IF

The conditional expression if(4) is converted to a Boolean value: non-zero values get converted to Boolean true, and zero-values get converted to Boolean false

24. Char

char8_t has been added in C++20
'x' - ASCII character to be enclosed in single quotes
"xxx" - strings to be enclosed in double quotes
Use \', \", \\, \? To print these characters
Char a = '\x4E' is nothing but ASCII N with decimal value 78 => char a = 78;

25. String literals

#include<string>, compound type,
To read a full line of input into a string, you're better off using the std::getline(std::cin >> std::ws, name); std::cin breaks at whitespace, takes '\n'
Use the std::ws input manipulator, to tell std::getline() to ignore any leading whitespace characters

26. Literal constants

A **literal** (also known as a **literal constant**) is a fixed value that has been inserted directly into the source code

integer, float, string, hex, binary, scientific notation.. Pay attention to suffixes.. 52l denotes long int.. 52u denotes unsigned, 52ul unsigned long

Literal value	Examples
integral value	5, 0, -3
boolean value	true, false
floating point value	3.4, -2.2
char value	'a'
C-style string	"Hello, world!"

Data Type	Suffix	Meaning
int	u or U	unsigned int
int	l or L	long
int	ul, uL, Ul, UL, lu, lU, Lu, or LU	unsigned long
int	ll or LL	long long
int	ull, uLL, Ull, ULL, llu, llU, LLu, or LLU	unsigned long long
double	f or F	float
double	l or L	long double

By default floating point literal constants have a type of *double*.

27. Symbolic constants

Symbolic constant is a name given to a constant literal value/magic number

Any variable that should not be modifiable after initialization and whose initializer is known at compile-time should be declared as constexpr.
Any variable that should not be modifiable after initialization and whose initializer is not known at compile-time should be declared as const.

Initializer known at compiletime -> compiletime constant
-> Use constexpr (better than #define - can be debugged like norm var, scope can be defined
-> Compiler compiles one translation unit at a time and sees constants available only in the source file being compiled i.e. extern'd const variables will be resolved at run-time and become run-time constants..
In order for variables to be usable in compile-time contexts, such as array sizes, the compiler has to see the variable's definition (not just a forward declaration) i.e. initialization
Initializer known at runtime -> runtime constant -> use const .. const int a{0} is compiletime and const int a{b} is runtime

28. Function arguments evaluation

C++ does not define the order of evaluation for function arguments or operator operands.

29. Conditional Ternary operator

Is most useful when we need a conditional initializer (or assignment) for a variable, or to pass a conditional value to a function
Ternary operator can be LHS too to

a. assign pointer or reference value based on condition.. **LHS should be l-value**

```
int* a = new int{};
int* b = new int{};
int c = 2;
*(1 ? a : b) = c; //l-value due to bracket
1 ? *a : *b = c;  // =, ?: are having same precedence.. So 1 ? *a : (*b = c);
 (1 ? a : b) = &c; //l-value

int c = 2;
int& a = c;
int& b = c;
((a==2) ? a : b) = 4;// LHS is l-value
(a==2) ? a : b = 4;  // LHS is r-value
```

b. Call function based on condition

30. Scope Resolution global namespace

::doSomething the identifier doSomething is looked for in the global namespace

31. Anonymous/unnamed namespaces and Inline namespaces

Anonymous namespace

- All content declared in an unnamed namespace is treated as if it is part of the parent namespace
- all identifiers inside an unnamed namespace are treated as if they had **internal linkage**
- typically used when you have a lot of content that you want to ensure **stays local to a given file**

Inline namespaces

- An inline namespace is a namespace that is typically used to version content.
- Much like an unnamed namespace, anything declared inside an inline namespace is considered part of the parent namespace. However, inline namespaces don't give everything internal linkage
- When called without namespace, objs within inline namespace is referred.. This preserves the function of existing programs while allowing newer programs to take advantage of newer/better variations.

```
namespace v2 // declare a normal namespace named v1
{
    void doSomething()
    {
        std::cout << "v2\n";
    }
}
inline namespace v1 // declare an inline namespace named v1
{
    void doSomething()
    {
        std::cout << "v1\n";
    }
}

doSomething(); // calls the inline version of doSomething() (which is v1)
v2::doSomething();
```

32. namespace aliases and nested namespaces

namespace a = foo::goo //alias
namespace foo:goo //nested namespace

33. Using directive and declaration (used for obj and namespace resp.)

qualified name - with scope
unqualified name - without scope

using declaration - declare an unqualified name as an alias of qualified name **using std::cout** .. Restricted to local or file scope
using directive (using namespace x) - import all the identifiers of namespace into scope .. **using namespace std** Can cause future collisions .. Restricted to local or file scope

Avoid using directive

34. Inline

use the inline keyword to define an external linkage const namespace scope variable, or any static class data member (inline static), in a header file, so that the multiple definitions that result when that header is included in multiple translation units are OK with the linker – it just chooses one of them
https://stackoverflow.com/questions/38043442/how-do-inline-variables-work

35. Scope, duration, linkage

- Scope **-SEE-** says where a single declaration can be seen and used (local/block, namespace, file, global)
- Duration **-LIFE-** (automatic, static, dynamic)
- Linkage -**ACCESS**- says whether multiple declaration in two locations in program can refer to same object or not
  - No - one declaration restricted to scope
  - Internal - declaration internal to file in which its declared (not exposed to linker)
  - External - accessible via different declaration outside the file where it is defined

Define variables in the most limited **existing** scope. Avoid creating new blocks whose only purpose is to limit the scope of variable

36. Local variables (scope, duration, linkage)

Blockscope - which means they are in scope from their point of definition to the end of the block they are defined within
Automatic duration - hence automatic variables
Linkage - no linkage (means 2 declaration in different scope access diff variables)

37. Global variables (avoid global scope/filescope worst case use const/constexpr in a UD-Namespace)

- Simple Globals
  - Global Scope
  - Static duration - start to end of program
  - Linkage - External (means external access of same var possible) .. Use extern keyword in other declarations
- Static Globals
  - File Scope
  - Static duration

        - Internal linkage
    - Const globals
        - File Scope or Global Scope
        - Internal Linkage by default (can be given external linkage by extern, **inline**)
          Rationale: Because const objects can be used as compile-time values in C++, this feature urges programmers to provide explicit initializer values for each const. This feature allows the user to put const objects in header files that are included in many compilation units.

    - Constexpr Globals
        - File Scope
        - Internal Linkage (can be given external linkage **inline**)
    - Functions
        - Global Scope
        - External linkage

Like global keyword in python to refer to global var, ::var can be used in local scope to refer to global var despite another var defined in local scope
Avoid **dynamic initialization** of global variables i.e. runtime initialization.. Use **static initialization** i.e. compiletime initialization

38. Extern keyword - give external linkage or forward declare

extern const int a = 0 ; //definition of const with external linkage
extern int a; // forward declaration - a defined elsewhere
extern const int a; // forward declaration - defined elsewhere

39. Constexpr have to be initialized during any declaration -> cannot be forward declared

Hence extern keyword is useless for constexpr

40. Use inline constexpr (C++17)

Linker will consolidate all inline definitions of a variable into a single variable definition (thus meeting the one definition rule).

41. Static keyword - make only one instance

a. Static globals have file-scope, internal linkage
   Static const globals or static constexpr globals

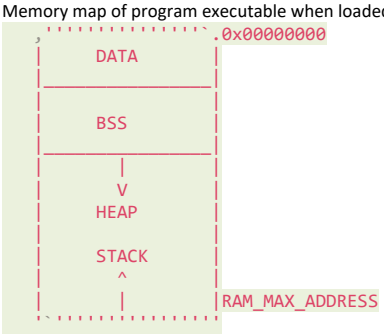
Const = literal const	**Static const**	**Constexpr**	**Static constexpr**
File Scope	File Scope	File-Scope	File-Scope
Internal Linkage	Internal linkage	Internal Linkage	Internal Linkage
Static duration	Static duration	Static duration	Static duration
Initialized in compiletime	Initialized in compiletime	Initialized in compiletime	Initialized in compiletime
Stored in .rodata	Stored in .rodata	Stored in .rodata	Stored in .rodata

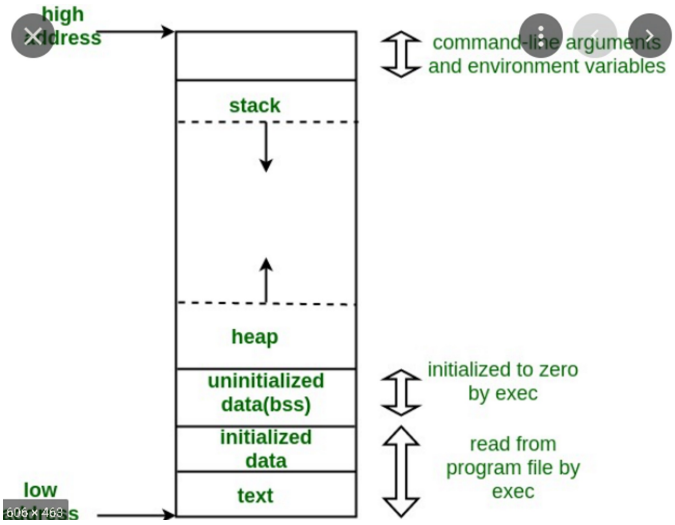
   Local static const/constexpr useful
b. Static local changes duration of var from automatic to static - start to end of program. So stored in .DATA/.BSS instead of stack. Scope and linkage remains

c. Uninitialized / Zero init static data goes in .BSS (Block Started by Symbol) and are zero-init by exec, initialized non-zero static goes in .DATA and are read by exec

42. ELF contents and memory map of loaded program

ELF -> https://upload.wikimedia.org/wikipedia/commons/e/e4/ELF_Executable_and_Linkable_Format_diagram_by_Ange_Albertini.png
Memory map of program executable when loaded





43. .TEXT

The code segment (also called a text segment), where the compiled program sits in memory. The code segment is typically read-only.

44. .DATA and .BSS

Data used by a program can be split into

- statically allocated global data regions - the **initialized** DATA segment and the **zero-initialised** BSS segment and
  - .DATA has read-only and read/write segments
- the runtime allocated memory - the Stack and the Heap.

Uninitialized globals and zero init globals goes in .BSS (Block Started by Symbol) and are init by exec to 0, initialized non-zero global vars goes in .DATA and are read by exec

Variables to be zero-initialized are placed in the .bss segment of the program image, which occupies no space on disk and is zeroed out by the OS when loading the program

Initial values of the non-zero vars of program which goes into .DATA is stored in the program exe (only if non-zero.. Zero not saved to save flash space)

45. STACK and HEAP

HEAP - keeps track of memory used for dynamic memory allocation
CALL STACK -

- keeps track of all the active functions (those that have been called but have not yet terminated)
- "items" we're pushing and popping on the stack are called stack frames. A stack frame keeps track of all of the data associated with one function call.
- SP keeps track of where the top of the call stack currently
- Memory used by Stack frames being popped off is not cleared hence the garbage value of locals
- Stack overflow happens when all the memory in the stack has been allocated

Function call

a. The program encounters a function call.
b. A stack frame is constructed and pushed on the stack.
   i. The address of the instruction beyond the function call (called the return address).
   ii. All function arguments.
   iii. Memory for any local variables.
   iv. Saved copies of any registers modified by the function that need to be restored when the function returns
c. The CPU jumps to the function's start point.

function termination:

a. Registers are restored from the call stack
b. The stack frame is popped off the stack. This frees the memory for all local variables and arguments.
c. The return value is handled.
d. The CPU resumes execution at the return address.

46. Storage Class (duration, linkage) specifier

C++ supports 4 active storage class specifiers:

Specifier	Meaning	Note
extern	static (or thread_local) storage duration and external linkage	
static	static (or thread_local) storage duration and internal linkage	
thread_local	thread storage duration	Introduced in C++11
mutable	object allowed to be modified even if containing class is const	

47. Inline function has to be defined in every translation unit its used - why?

going to replace call with func content so?

48. Implicit conversions

implicit type conversion: promotions (smaller to larger in below order) and conversions(larger to smaller or different types)

a. long double (highest)
b. double
c. float
d. unsigned long long
e. long long
f. unsigned long
g. long
h. unsigned int
i. int (lowest)

49. Enum - distinct type whose value is restricted to a range of values

- Enum types are considered part of the integer family of types, and it's up to the compiler to determine how much memory to allocate for an enum variable. The C++ standard says the enum size needs to be large enough to represent all of the enumerator values. Most often, it will make enum variables the same size as a standard int.
- **Defining enum doesn't allocate memory.. When a variable of its type is created only then memory is allocated**

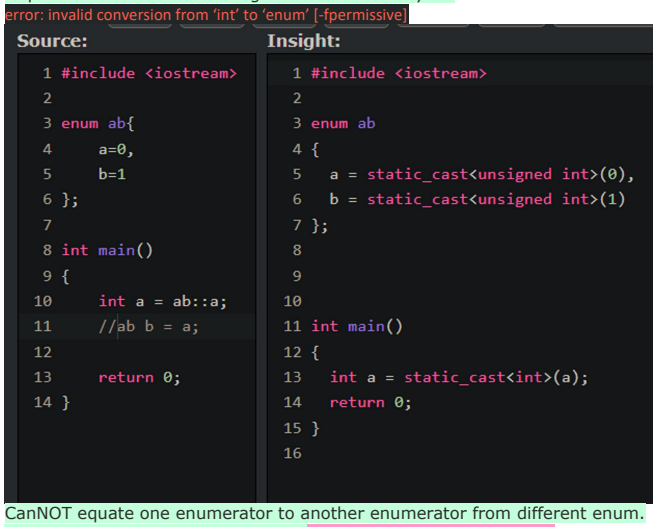
Unscoped enumeration

```
enum name(optional) { enumerator = constexpr , enumerator = constexpr , ... }          (1)
enum name(optional) : type { enumerator = constexpr , enumerator = constexpr , ... }   (2)   (since C++11)
enum name : type ;                                                                      (3)   (since C++11)
```

1) Declares an unscoped enumeration type whose underlying type is not fixed (in this case, the underlying type is an implementation-defined integral type that can represent all enumerator values; this type is not larger than `int` unless the value of an enumerator cannot fit in an `int` or `unsigned int`. If the *enumerator-list* is empty, the underlying type is as if the enumeration had a single enumerator with value 0).
2) Declares an unscoped enumeration type whose underlying type is fixed.

Simple Enum

- Enumerators (constants in enum type) scope is same as enumeration, Placed in the same namespace as enumeration

- Anonymous enums allowed
  ```
  enum {
  maxAntNum = 4;
  }
  ```
- Implicit cast for enum -> integer but not otherway …
  error: invalid conversion from 'int' to 'enum' [-fpermissive]
- 

- CanNOT equate one enumerator to another enumerator from different enum.
- Can compare one enum to another - but with warning by compiler
- enum Color : std::uint8_t … during forward declaration this is needed

50. Enum class is type-safe i.e. it's an entirely new type + enumerators in it are scoped too

- Enumerators are placed in enum type's namespace and its placed in enum's scope
- Anonymous 'scoped enum' is not allowed
- Explicit cast needed to print value eg static_cast. No Implicit casts
  Static_cast can be used to obtain value of enum class.
- Cannot equate one enumerator to another enumerator from different enum..
- Relational operators cannot be used for two different classes

Scoped enumerations

`enum struct|class` *name* { *enumerator* = *constexpr* , *enumerator* = *constexpr* , … }	(1)
`enum struct|class` *name* : *type* { *enumerator* = *constexpr* , *enumerator* = *constexpr* , … }	(2)
`enum struct|class` *name* ;	(3)
`enum struct|class` *name* : *type* ;	(4)

1) declares a scoped enumeration type whose underlying type is `int` (the keywords `class` and `struct` are exactly equivalent)
2) declares a scoped enumeration type whose underlying type is *type*
3) opaque enum declaration for a scoped enumeration whose underlying type is `int`
4) opaque enum declaration for a scoped enumeration whose underlying type is *type*

Each *enumerator* becomes a named constant of the enumeration's type (that is, *name*), which is contained within the scope of the enumeration, and can be accessed using scope resolution operator. There are no implicit conversions from the values of a scoped enumerator to integral types, although `static_cast` may be used to obtain the numeric value of the enumerator. (since C++11)

51. Enum initialization using using list initialization

Scoped Enums can be initialized from integer below way using list initialization

```
enum class X : int {};
X xenum {42};
X xenum = {42}; // NOK
X xenum = X{42}; // OK
```

Only allowed when X type is fixed, **conversion is non-narrowing**, initializer list has just one element.

52. std::size(..)

The std::size() function from the <iterator> header can be used to determine the length of arrays, std::array, std::vector

53. Use initializer list to initialize array

```
int prime[5]{ 2, 3, 5, 7, 11 }; // use initializer list to initialize the fixed array
```

54. Array as function arguments

Copying large arrays can be very expensive, C++ does not copy an array when an array is passed into a function. Instead only pointer to array is passed - note below. This has the side effect of allowing functions to directly change the value of array elements!

```
void passArray(int prime[5])
```

NOTE - When passed to function fixed array will decay (be implicitly converted) into a pointer that points to the first element of the array

Array can be passed by reference too

```
void func(int (&arr)[2])
```

55. Types of array declaration

```
int prime[5]{ 2, 3, 5, 7, 11 };    // A2_i .. Type -> int[5]
```

```
int (*prime)[5]{2, 3, 4, 5, 6};     // PA2_i .. type -> int (*) [5] -> sizeof(int(*)[5])) is 5*4
int* prime[5]{ 2, 3, 5, 7, 11 };   // A2_P
int (&prime)[5]{ 2, 3, 5, 7, 11 }; // A2_i
```

56. Array declaration types

int[5], int(*)[5],int* -- need conversion among these vars… but same var can be used to access like arr[2]

```
// polymorphic -> behaves like pointer and array.
// arr prints address of array[0] i.e arr behaves as pointer, arr type though is int[5]
int arr[5] = {};
std::cout << typeid(arr).name() << "," << &arr[0] << "," << arr << std::endl;
A5_i int[5], 0x7ffe4cf9b320, 0x7ffe4cf9b320

// polymorphic -> arr2 means pointer to array[0] elem i.e. int(*)[5], but arr2[0] can be used to access element like its int[5]
int (*arr2)[5] = static_cast<int(*)[5]>(&arr); //this is possible because &arr is of type PA5_i
std::cout << typeid(arr2).name() << "," << &arr2[0] << "," << arr2 << std::endl;
PA5_i int(*)[5], 0x7ffe4cf9b320, 0x7ffe4cf9b320

int (*arr1)[5] = new int[3][5];
std::cout << typeid(arr1).name() << "," << &arr1[0][0] << "," << &arr1[0] << "," << arr1[0] <<  "," << arr1 << std::endl;
PA5_i int(*)[5], addr of first elem: 0x2490c20, addr of first array: 0x2490c20, addr in first int* array: 0x2490c20, 0x2490c20
```

57. Converting int[5] to int(*)[5]

```
int arr[5] = {};
std::cout << typeid(arr).name() << "," << &arr[0] << "," << arr << std::endl;

int (*arr2)[5] = static_cast<int(*)[5]>(arr); // Cannot convert int[5] to int(*)[5]

int (*arr2)[5] = static_cast<int(*)[5]>(&arr);
int (*arr2)[5] = &arr;
std::cout << typeid(arr2).name() << "," << &arr2[0] << "," << arr2 << std::endl;
```

58. new and delete for array

//std::nothrow - tell new to return a null pointer if memory can't be allocated. This is done by adding the constant std::nothrow between the new keyword and the allocation type

```
int* a {new (std::nothrow) int{6}}
int* b {new (std::nothrow) int[2]{6,7}}[
int (*c)[5] = new int[3][5] //Type is int (*) [5] --> c is a pointer to first element
auto d = new int[3][5]
delete a;
delete [] b; //at array new[] keeps track of how much memory was allocated to a variable, so that array delete[] can delete properly
```

Set pointers to nullptr after delete to avoid dangling pointer.
Memory leak can be avoided if u don't lose start of alloc memory. Also delete alloc memory at the end of pointer scope.

59. Pass-by-value is the best way to accept parameters of **fundamental types** most cases

I

60. C++ really passes everything by value.. Pointers also. References also (actually pointers)

I

61. Do not function return by addr/reference for its local variables..

Use it for variables which are in scope (outer scope) in caller or use it for returning literals.

62. Overloaded Function Matching

1) First, C++ tries to find an exact match of args and types
2) If no exact match is found, C++ tries to find a match through promotion -> int to unsigned not allowed
3) If no promotion is possible, C++ tries to find a match through standard conversion.
   - Any numeric type will match any other numeric type, including unsigned (e.g. int to unsigned int, int to float)
   - Enum will match the formal type of a numeric type (e.g. enum to float)
   - Zero will match a pointer type and numeric type (e.g. 0 to char*, or 0 to float)
   - A pointer will match a void pointer
4) Finally, C++ tries to find a match through user-defined conversion.

63. default arguments of function

All default arguments must be for the rightmost parameter in function definition
Optional parameters do NOT count towards the parameters that make the function unique i.e. func(int), func(int, int=1) isn't overloaded func

Optional args' default values are resolved at compile-time!!!

64. In general we use (*name) in pointer decl/def because

type* name[] implies array of pointers of type ---------- while type (*name)[] implies pointer to array of type = array name
type* name() implies func which returns pointer to type --------- while type (*name)() implies pointer to func with return type=type

65. Function pointer syntax

Function pointer type -> <returntype> (*) (inputarg1type, …)

```
int func(){…}; //Just func refers to pointer, &func is same
int (*ptr)(){&func};
int (*ptr)(){};
ptr = &func;
(*ptr)(); // -----> function call
```

C++ implicit conversion of a function into a function pointer

```
int (*ptr)(){func};
ptr = func;
```

C++ implicit dereference of functionPtr
```
ptr(); // ------> function call
```

With function pointers all default args shud be specified before compilation!!! -----> because pointers resolve/bind to obj in run-time

```
bool(*fn_pt)(char,char) = fncomp;
std::map<char,int,bool(*)(char,char)> fifth (fn_pt); // function pointer as Compare
```

66. callback function syntax

Lets say
```
bool ascending(int x, int y)
```
Pointer to this function would be
```
bool (*comparisonFcn)(int, int);
```
Callback can be defined as
```
void selectionSort(int *array, int size, bool (*comparisonFcn)(int, int))
```
OR
```
void selectionSort(int *array, int size, bool comparisonFcn(int, int))
```
OR
```
void selectionSort(int *array, int size, bool (*comparisonFcn)(int, int) = ascending);
```
OR
```
using ValidateFunction = bool(*)(int, int);
bool validate(int x, int y, ValidateFunction pfcn)
```
OR
```
bool validate(int x, int y, std::function<bool(int, int)> fcn);
```

67. Attributes

are kind of a C++ hack. They're a solution to meta-problems with the evolution of the C++ language.

Attribute are a modern C++ feature that allows the programmer to
a. provide the compiler with some additional data about the code.
b. Suppress compiler warnings
c. Provide constraints for eg [[expects: i > 0]] ... Violation of the contract results in invocation of violation handler or if not specified then std::terminate()

To specify an attribute, the attribute name is placed between double hard braces.
```
    case 2:
        std::cout << 2 << '\n'; // Execution begins here
        [[fallthrough]]; // intentional fallthrough -- note the semicolon to indicate the null statement
    case 3:
        std::cout << 3 << '\n'; // This is also executed
        break;
```

https://en.cppreference.com/w/cpp/language/attributes
https://gcc.gnu.org/onlinedocs/gcc/Attribute-Syntax.html#Attribute-Syntax

Use: https://stackoverflow.com/questions/62856209/attributes-in-c-and-their-use

Standard attributes

Only the following attributes are defined by the C++ standard.

[[noreturn]] (C++11)	indicates that the function does not return
[[carries_dependency]] (C++11)	indicates that dependency chain in release-consume std::memory_order propagates in and out of the function
[[deprecated]] (C++14) [[deprecated("reason")]] (C++14)	indicates that the use of the name or entity declared with this attribute is allowed, but discouraged for some *reason*
[[fallthrough]] (C++17)	indicates that the fall through from the previous case label is intentional and should not be diagnosed by a compiler that warns on fall-through
[[nodiscard]] (C++17) [[nodiscard("reason")]] (C++20)	encourages the compiler to issue a warning if the return value is discarded
[[maybe_unused]] (C++17)	suppresses compiler warnings on unused entities, if any
[[likely]] (C++20) [[unlikely]] (C++20)	indicates that the compiler should optimize for the case where a path of execution through a statement is more or less likely than any other path of execution
[[no_unique_address]] (C++20)	indicates that a non-static data member need not have an address distinct from all other non-static data members of its class
[[optimize_for_synchronized]] (TM TS)	indicates that the function definition should be optimized for invocation from a synchronized statement

68. Switch rules

a. All statements below case belong to switch block (the only block)
b. can declare (but not initialize) variables inside the switch, both before, after & inside the case labels.. Use {} in case to define variable

69. C++ halts ---- #include <cstdlib>

```
main()
{
    return 0; //std::exit(status) is called
}
```

a. std::exit(statusCode) is a function that causes the program to terminate normally. Normal termination means the program has exited in an expected way. Performs cleanup of static object before returning control to OS. Does not clean up local variables in the current function or cleanup the call stack.
If the program terminates (via std::exit) then we will have lost our call stack and any debugging information that might help us isolate the problem

b. std::abort() function causes your program to terminate abnormally. Abnormal termination means the program had some kind of unusual runtime error and the program couldn't continue to run.

c. std::terminate() function is typically used in conjunction with exceptions. Called implicitly when an exception isn't handled.
std::terminate() calls std::abort()

I

70. Ellipsis operator

Ij

71. Const & volatile type qualifiers

For any type T (including incomplete types), other than function type or reference type, there are three more distinct types in the C++ type system: *const-qualified* T, *volatile-qualified* T, and *const-volatile-qualified* T.

a. const object - an object whose type is const-qualified, or a non-mutable subobject of a const object. Attempt to modify it will cause undefined behavior.
b. volatile object - an object whose type is volatile-qualified, or a subobject of a volatile object, or a mutable subobject of a const-volatile object.
Attempt to access volatile object thru a glvalue of non-volatile type causes undefined behavior.

Calling non-volatile member function via volatile object results in passing 'volatile B' as 'this' argument discards qualifiers.. So this arg is volatile and function is not and doesn't expect volatile this..
This is fixed by making function volatile i.e. volatile keyword after function name
Opposite i.e. calling volatile member function from non-volatile obj works but may result in undefined behavior

c. cv

- ***const volatile object*** - an object whose type is *const-volatile-qualified*, a non-mutable subobject of a const volatile object, a const subobject of a volatile object, or a non-mutable volatile subobject of a const object. Behaves as both a const object and as a volatile object.

From https://en.cppreference.com/w/cpp/language/cv

References and pointers to cv-qualified types may be implicitly converted to references and pointers to *more cv-qualified* types. In particular, the following conversions are allowed:

- reference/pointer to *unqualified* type can be converted to reference/pointer to `const`
- reference/pointer to *unqualified* type can be converted to reference/pointer to `volatile`
- reference/pointer to *unqualified* type can be converted to reference/pointer to `const volatile`
- reference/pointer to `const` type can be converted to reference/pointer to `const volatile`
- reference/pointer to `volatile` type can be converted to reference/pointer to `const volatile`

Note: additional restrictions are imposed on multi-level pointers.

To convert a reference or a pointer to a cv-qualified type to a reference or pointer to a *less cv-qualified* type, const_cast must be used.

72. Mutable

mutable - permits modification of the class member declared mutable even if the containing object is declared const.

73. Volatile

a. C and C++ give you the chance to explicitly disable caching of variables in registers. If you use the volatile modifier on a variable, the compiler won't cache that variable in registers — each access will hit the actual memory location of that variable.

b. Unlike const, volatile discriminates between primitive types and user-defined types. Namely, unlike classes, primitive types still support all of their operations (addition, multiplication, assignment, etc.) when volatile-qualified.
For example:
   i. you can assign a non-volatile int to a volatile int, but you cannot assign a non-volatile object to a volatile object i.e. I think u cannot take a volatile obj and assign it to a non-volatile obj.

   ii. Non-volatile obj to volatile conversion is allowed

   iii. Opposite only via const_cast

```
Gadget& ref = const_cast<Gadget&>(volatileGadget);
```

c. Just like constness, volatileness propagates from the class to its members i.e. objs declared volatile -> its members are volatile too.. However volatile objs cant call non-volatile member functions
error: passing 'volatile AClass' as 'this' argument discards qualifiers [-fpermissive]

Useful how? -> when you declare obj as volatile, for eg vector, calling non-volatile member func is banned. I.e. volatile obj must be converted to non-volatile and only then its non-volatile member functions can be accessed i.e. in a critical section after mutex locking or in single threaded env

d. Data shared between threads is conceptually volatile outside a critical section, and non-volatile inside a critical section. You enter a critical section by locking a mutex. You remove the volatile qualifier from a type by applying a const_cast.

From <https://www.drdobbs.com/cpp/volatile-the-multithreaded-programmers-b/184403766>

74. Const volatile - code shouldn't change it but hardware may. Or some external force ;)

How to use const and volatile together

Though the essence of the volatile ("ever-changing") and const ("read-only") keywords may seem at first glance opposed, they are in fact orthogonal. Thus sometimes it makes sense to use both keywords in the declaration of a single variable or pointer. Typical use scenarios for both involve either pointers to memory-mapped hardware registers or variables located in shared memory areas.

75. Malloc - contiguous memory. Not initialized.

Calloc - contiguous memory blocks. Initialized.
Realloc - reallocate above memory. Content of the memory block is preserved up to the lesser of the new and old sizes, even if the block is moved to a new location. If the new size is larger, the value of the newly allocated portion is indeterminate. If ptr given is null, then malloc is done.

From <https://www.cplusplus.com/reference/cstdlib/realloc/>

76. #include <initializer_list>

Compiler will automatically construct a non-empty object of this class template type whenever an initializer list expression needs to be passed or copied.

```
std::initializer_list<int> mylist;
mylist = { 10, 20, 30 };
for (int x: mylist) std::cout << ' ' << x;

mylist.begin();//pointer to first element
mylist.end();  //pointer to last+1 element
mylist.size(); //number of elements of initializer_list
```

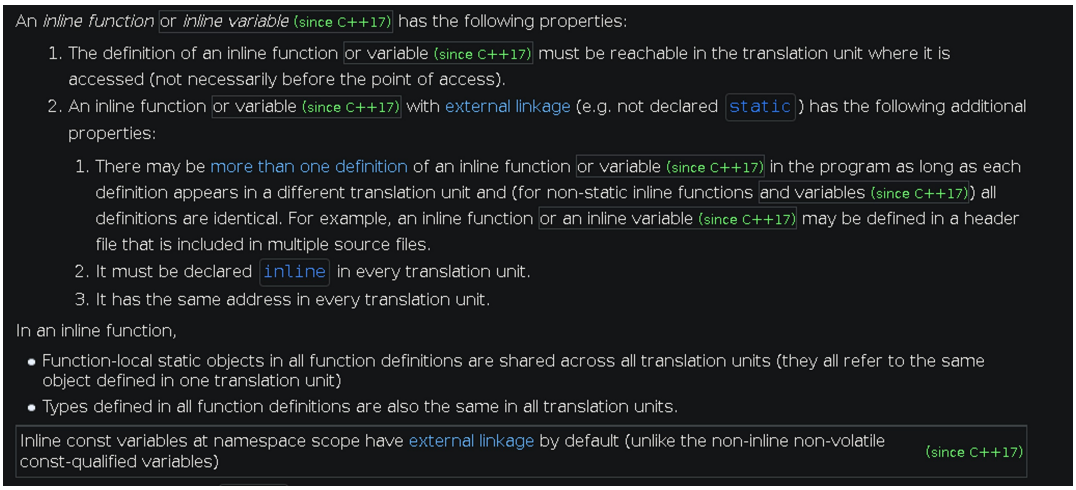
77. Range based for-loop (C++11)

```
std::vector<int> v = {0, 1, 2, 3, 4, 5};

for (const int& i : v) // access by const reference
    std::cout << i << ' ';
std::cout << '\n';

for (auto i : v) // access by value, the type of i is int
    std::cout << i << ' ';
std::cout << '\n';

for (auto&& i : v) // access by forwarding reference, the type of i is int&
    std::cout << i << ' ';
std::cout << '\n';

const auto& cv = v;

for (auto&& i : cv) // access by f-d reference, the type of i is const int&
    std::cout << i << ' ';
std::cout << '\n';

for (int n : {0, 1, 2, 3, 4, 5}) // the initializer may be a braced-init-list
    std::cout << n << ' ';
std::cout << '\n';

int a[] = {0, 1, 2, 3, 4, 5};
for (int n : a) // the initializer may be an array
    std::cout << n << ' ';
std::cout << '\n';
```

https://en.cppreference.com/w/cpp/language/range-for

78. Inline

https://en.cppreference.com/w/cpp/language/inline#:~:text=The%20inline%20specifier%2C%20when%20used%20in%20a%20decl%2Dspecifier%2D,is%20implicitly%20an%20inline%20variable.



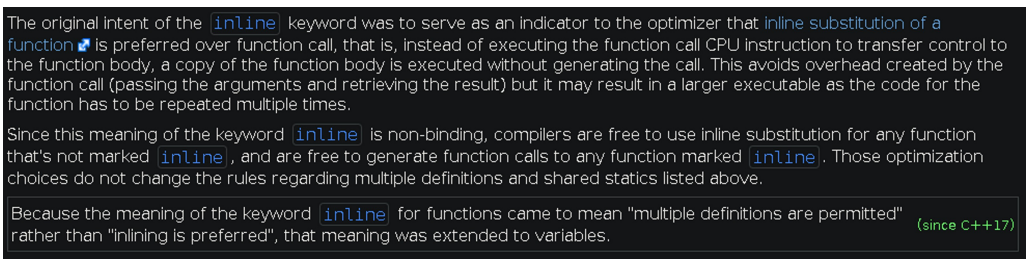
An *inline function* or *inline variable* (since C++17) has the following properties:

1. The definition of an inline function or variable (since C++17) must be reachable in the translation unit where it is accessed (not necessarily before the point of access).
2. An inline function or variable (since C++17) with external linkage (e.g. not declared `static`) has the following additional properties:
    1. There may be more than one definition of an inline function or variable (since C++17) in the program as long as each definition appears in a different translation unit and (for non-static inline functions and variables (since C++17)) all definitions are identical. For example, an inline function or an inline variable (since C++17) may be defined in a header file that is included in multiple source files.
    2. It must be declared `inline` in every translation unit.
    3. It has the same address in every translation unit.

In an inline function,

- Function-local static objects in all function definitions are shared across all translation units (they all refer to the same object defined in one translation unit)
- Types defined in all function definitions are also the same in all translation units.

Inline const variables at namespace scope have external linkage by default (unlike the non-inline non-volatile const-qualified variables) (since C++17)



The original intent of the `inline` keyword was to serve as an indicator to the optimizer that inline substitution of a function is preferred over function call, that is, instead of executing the function call CPU instruction to transfer control to the function body, a copy of the function body is executed without generating the call. This avoids overhead created by the function call (passing the arguments and retrieving the result) but it may result in a larger executable as the code for the function has to be repeated multiple times.

Since this meaning of the keyword `inline` is non-binding, compilers are free to use inline substitution for any function that's not marked `inline`, and are free to generate function calls to any function marked `inline`. Those optimization choices do not change the rules regarding multiple definitions and shared statics listed above.

Because the meaning of the keyword `inline` for functions came to mean "multiple definitions are permitted" rather than "inlining is preferred", that meaning was extended to variables. (since C++17)