Introduction to Standard C++

Lecture 01: The C core in C++

Massimiliano Culpo[1]

[1]CINECA - SuperComputing Applications and Innovation Department

07.04.2014

Table of contents - Language rules

Table of contents - Language rules

C++ 101: the "Hello world!"

```
// 1 - Preprocessor directive
#include <iostream>

// 2 - Using directive on a  namespace
using namespace std;

int main() { // 3 - Program entry point
   cout << "Hello world!" << endl;
   return 0;
}
```

The **main** function (§3.6.1)

All implementations shall allow:

```
int main() {
  /*...*/
}
```

and:

```
int main(int argc, char * argv[]) {
  /*...*/
}
```

1. **argc** is the number of arguments (non-negative)
2. if **argc** is non-zero the arguments are supplied through **argv**
3. **argv**[0] is a pointer to the name used to invoke the program
4. **argv**[**argc**] shall be 0

Declarations and definitions (§3.1)

1. Every name that denotes an entity is introduced by a declaration
2. A declaration may:
   - introduce one or more names into a translation unit
   - re-declare names introduced by previous declarations
3. A declaration is a definition unless:
   - it declares a function without specifying the function body
   - it contains the **extern** specifier and neither an initializer nor a function-body
   - it declares a static data member in a class definition
   - it is a class name declaration or a typedef declaration
   - it is a using declaration or a using directive

One Definition Rule (§3.2.1)

No translation unit shall contain more than one definition of any variable, function, class type, enumeration type, or template.

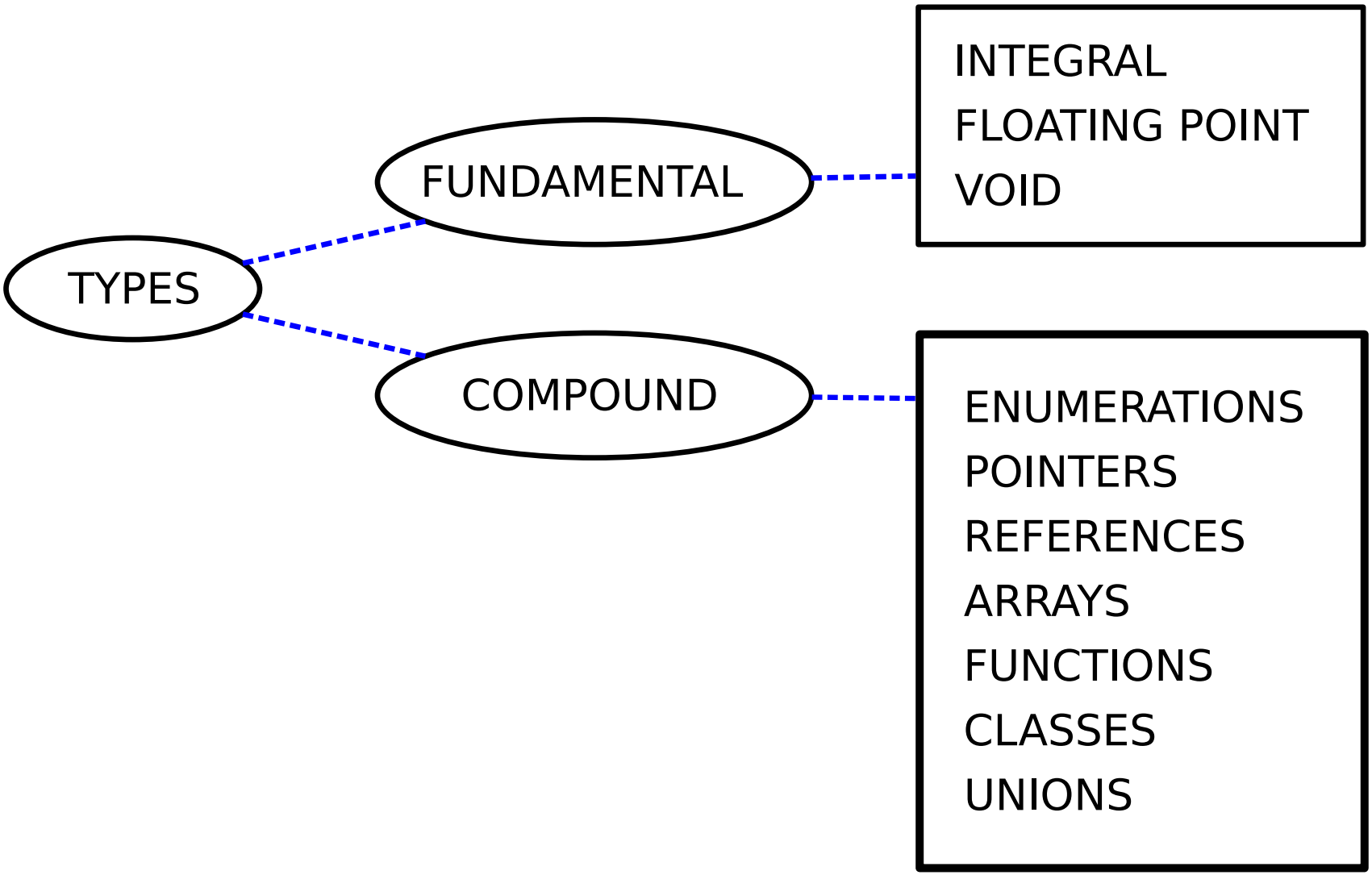
Q: What is a declaration/definition? (§3.1.2)

```
int a;
extern const int c = 1;
int f(int x) {
  return x+a;
}
struct X {
  int x;
};

extern int a;
extern const int c;
int f(int);
typedef int Int;
using N::d;
```

Table of contents - Language rules

Taxonomy of types (§3.9)



Fundamental types (§3.9.1)

Integral types are composed of **bool** and:

Signed integer types

- **signed char**
- **short int**
- **int**
- **long int**
- **long long int** (C++ 11)

Unsigned integer types

- **unsigned char**
- **unsigned short int**
- **unsigned int**
- **unsigned long int**
- **unsigned long long int** (C++ 11)

Floating-point types include **float**, **double** and **long double**

The void type is an incomplete type and has an empty set of values

Compound types: enumerations (§7.2)

An enumeration is a distinct type with named constants:

```
enum {
   yellow = 10
   red,
   blue,
   black = blue + 2
};
```

```
// C++11 only
enum class State :
unsigned int {
   On,
   Off
};
```

1. The identifiers in an enumerator-list are declared as constants
2. An enumerator-definition without an initializer:
   - increase the value of the previous enumerator by one
   - for the first enumerator the value of the corresponding constant is zero
3. Strongly typed enum (C++ 11): no implicit conversion

Compound types: pointers (§8.3.1)

A pointer variable stores the address of another variable:

```
int    i   = 10;
int  * pi  = &i;

*pi = 20;
```

1. The unary operator * performs indirection (§5.3.1):
   - shall be applied to a pointer
   - returns an lvalue referring to the object or function
2. The result of the unary operator & is a pointer to its operand
3. A pointer to an incomplete type cannot be dereferenced
4. The keyword **nullptr** indicates a null pointer (C++ 11)

Compound types: references (§8.3.2)

A reference can be thought of as an alias name for an object:

```
int    i   = 10;
int & ri   = i;

ri = 20; // Now i == 20 holds true
```

1. It combines the syntax of values with the semantic of pointers
2. It is unspecified whether or not a reference requires storage
3. According to the standard you can't declare:
   - references to references
   - pointers to references
   - arrays of references
4. A reference shall be initialized to a valid object or function

Compound types: arrays (§8.3.4)

An array contains a contiguously allocated, non-empty set of objects:

```
int values[3];

for (int ii = 0; ii < 3; ++ii)
  values[ii] = 10*ii;
```

1. The number of elements is specified by a constant integral expression
2. If the constant integral expression:
   - has value N, the array has N elements numbered $[0, N-1]$
   - is omitted, the type of the identifier of D is an incomplete object type
3. An array can't be constructed from **void** or references

Compound types: arrays (§8.3.4)

Several adjacent **array of** specifications declare a multidimensional array:

```
int    imarray[30][10];
float  fmarray[10][20][30];

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

1. Arrays in C++ are stored row-wise (last subscript varies fastest)
2. Only the first constant expression may be omitted:
   - where an incomplete object type is allowed
   - where an initializer is used

Compound types: functions (§8.3.5)

A function is the typical way to accomplish a task in C++:

```
int    next_element();
float  square_root(float a );
double square_root(double a);
```

1. If the parameter list is empty, the function takes no arguments
2. The order of evaluation of function arguments is unspecified
3. A single name can be used for several different functions in a single scope (function overloading §13)

Compound types: functions (§8.3.6)

Functions may use default arguments in their parameter declaration:

```
int max(int a, int b = 0, int c = 0) {
  int m = (a > b) ? a : b;
  return (m > c) ? m : c;
}
```

1. Default arguments are evaluated each time the function is called
2. A default argument shall not be redefined by a later declaration
3. Nested function definitions are not allowed in C++

Q: What are the types of the following variables?

```
int i;
int *pi;
int f();
int *fpi(int);
int (*pif)(char*, char*);
int (*fpif(int))(int);

char  * a,b;
float *  a[10];
float (*a)[10];
int (*f ) (int, float*);
int (*&g) (int, float*)
```

Table of contents - Language rules

Scope (§3.3)

Each identifier is valid only within some possibly discontiguous portion of program text called its scope:

```
int j = 24;
int main() {
  int i = j, j;
  j = 42;
}
```

In the previous snippet the identifier **j** is declared twice as a name:

Scope of the "first" **j**

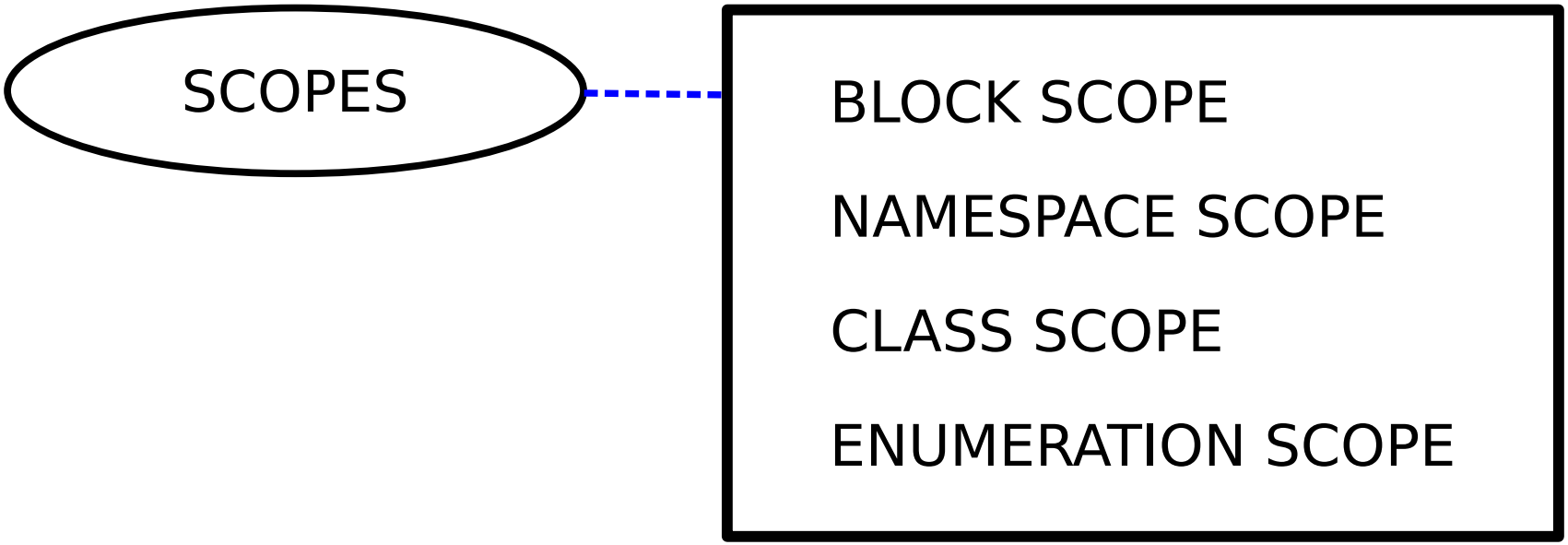
- begin after its declaration
- extends until the end of the program
- excludes the text between , and }

Scope of the "second" **j**

- begin after its declaration
- extends until }

The inner name hides the outer one

Taxonomy of scopes (§3.3)



Block scope (§3.3.3)

A portion of program text enclosed between { and } is called a block:

```
int a[10];
// Outer scope where ii is float
float ii = 999.0f;
for(int ii = 0; ii < 10; ii++) {
    a[ii] = ii; // ii is int
}
```

1. A name declared in a block is local to that block (it has block scope)
2. Its potential scope begins with its declaration and ends with the block
3. Names declared in the "condition part" of a statement are local to that statement

Namespace (§7.3) and namespace scope (§3.3.6)

A namespace is an optionally-named declarative region:

```
namespace nms {
  int counter() {
    /* ... */
  }
}
// Qualified name lookup
int id = nms::counter();
```

1. The name of a namespace can be used to access entities
2. Its definition can be split over different translation units
3. The outermost declarative region of a translation unit is a namespace, called the global namespace

Namespace (§7.3) and namespace scope (§3.3.6)

Namespace definitions can be nested:

```
namespace Outer {
  int i = 0;
  namespace Inner {
    void f() { i++; } // Outer::i
    int i = 10;
    void g() { i++; } // Inner::i
  }
}
```

An unnamed namespace behaves as if it were replaced by:

```
namespace unique { /* body */ }
```

Namespace (§7.3) and namespace scope (§3.3.6)

All occurrences of **unique** in a translation unit:

```
namespace { int i; } // unique ::i
void f()  { i++;   } // unique ::i++
namespace A {
  namespace {
    int i; // A:: unique ::i
    int j;
  }
  void g() { i++; } // A:: unique ::i++
}
```

1. are replaced by the same identifier...
2. ...that differ from all other identifiers in the entire program

Namespace (§7.3) and namespace scope (§3.3.6)

A namespace-alias (§7.3.2):

```
namespace long_long_name {
   int i = 0;
}

// Namespace alias
namespace lln = long_long_name;
// i has value 1
long_long_name::i++;
// i has value 2
lln::i++;
```

declares an alternate name for a namespace.

Using declarations (§7.3.3)

A using-declaration introduces a name into a declarative region:

```
namespace B {
  void f(char) { /* ... */ };
}

namespace D {
  using B::f;
  void f(int) {
    f('c'); // B::f(char)
  }
  void g(int) {
    g('c'); // D::g(int)
  }
}
```

Using declarations (§7.3.3)

The entity declared by a using-declaration:

```
namespace A {
  void f(int);
}
using A::f; // synonym for A::f(int);
namespace A {
  void f(char);
}
void foo() {
  f('a'); // calls f(int),
}         // even though f(char) exists.
```

shall be known in the context using it according to its definition at the point of the using-declaration.

Using directive (§7.3.4)

A using-directive may only appear in namespace scope or in block scope:

```
namespace A {
   void f(int);
   int  g(float);
}

using namespace A;

// void A::f(int)
f(10);
// int  A::g(float)
int a = g(3.0f);
```

Using directive (§7.3.4)

For unqualified lookup, the using-directive is transitive:

```
namespace M {
  int foo(char);
}
namespace N {
  int foo(float);
  // Import int M::foo(char)
  using namespace M;
}
void f() {
  using namespace N;
  int foo('a');  // int M::foo(char)
  int foo(1.0f); // int N::foo(float)
}
```

Table of contents - Language rules

Standard conversions (§4)

Standard conversions are implicit conversions defined for built-in types.

A standard conversion sequence is a sequence of standard conversions:

1. Zero or one conversion from the following set:
   - lvalue-to-rvalue conversion
   - array-to-pointer conversion
   - function-to-pointer conversion
2. Zero or one conversion from the following set:
   - integral promotions
   - floating point promotion
   - pointer conversions
   - pointer to member conversions
   - integral conversions
   - floating point conversions
   - floating-integral conversions
   - boolean conversions
3. Zero or one qualification conversion

It will be applied to an expression if necessary to convert it to a required destination type.

Standard conversions (§4)

1. An expression **e** can be implicitly converted to a type **T** if and only if:

```
T t = e;
```

   is well-formed. The effect of the implicit conversion is the same as using the temporary variable **t** as the result of the conversion

2. Expressions with a given type will be implicitly converted:
   - when used as operands of operators
   - when used in the condition of an if or iteration statement
   - when used in the expression of a switch statement
   - when used as the source expression for an initialization

Table of contents - Language rules

Linkage specifications (§7.5)

Linkage between C++ and non-C++ code fragments:

```
// C++ linkage by default
complex sqrt(complex);

extern "C" {
// C linkage: no overloading
double sqrt(double);
}
```

can be achieved using a linkage-specification.

1. The Standard specifies the semantics for "C" and "C++"
2. A linkage-specification shall occur only in namespace scope

Linkage specifications (§7.5): example

```
int x;
namespace A {
  extern "C" int f();
  extern "C" int g() { return 1; }
  extern "C" int h();
  // ill-formed: same name
  extern "C" int x();
}
namespace B {
  // A::f or B::f
  extern "C" int f();
  // ill-formed: two definitions
  extern "C" int g() { return 1; }
}
// A::h and ::h refer to the same function
int A::f() { return 98; } // definition
extern "C" int h() { return 97; } // definition
```

Table of contents - Best practices

Table of contents - Best practices

Include Guards

It is common practice to protect header files with a unique macro name called include guard:

```
#ifndef HEADER_UNIQUE_NAME_
#define HEADER_UNIQUE_NAME_
 /*
  Header body
 */
#endif
```

to prevent violations of the One Definition Rule when including them.

Q1: Why the include guard for a header file needs to be unique?
Q2: What will happen in case of a name-clash?

Table of contents - Best practices

Prefer consts or enums to object-like macros

When you do something like this:

```
#define ASPECT_RATIO 1.653
```

the symbolic name **ASPECT_RATIO** is not seen by the compiler, as it is removed by the preprocessor during macro expansion.

This may have unexpected side-effects as:

- macro doesn't respect any scope
- compiler messages may refer to 1.653 instead of **ASPECT_RATIO**

The solution is to replace the macro with a constant or an enum:

```
const double AspectRatio = 1.653;
const char * const language_name = "C++";
enum { NumTurns = 5 };
```

Prefer inlines to function-like macros

Another common misuse of the **#define** directive is using it to implement function-like macros that don't incur the overhead of a function call:

```
// call f with the maximum of a and b
#define CALL_WITH_MAX(a, b) \
f((a) > (b) ? (a) : (b))
```

Anyhow, the use of **inline** can get the same efficiency plus all the predictable behavior and type safety of a regular function:

```
inline void callWithMax(int a, int b) {
  f(a > b ? a : b);
}
```

Use const whenever possible

The keyword **const** allows to specify a semantic constraint (a particular object should not be modified):

```
char greeting[] = "Hello";
// non-const pointer, non-const data
char *p = greeting;
// non-const pointer, const data
const char *p = greeting;
// const pointer, non-const data
char * const p = greeting;
// const pointer,  const data
const char * const p = greeting;
```

This cause the compiler to know and enforce your intentions, and let other programmers to be aware of the object properties.

Table of contents - Best practices

Prefer **std::vector** to built-in arrays

Even though knowing the array machinery in C++ is somehow mandatory, **std**::**vector** provides many advantages over built-in arrays:

```
#include <vector>
#include <iostream>
using namespace std;
void print_size(const vector<int>& foo) {
  // Maintain size information
  cout << foo.size()<< endl;
}
int main() {
  vector<int> foo;
  for (int ii = 0; ii < 1000; ii++)
    foo.push_back(ii); // Automatic resizing
  cout << foo[0] << endl; // Array syntax
  cout << foo.at(10) << endl; // Range check
  print_size(foo);
}
```

The one slide summary of the lecture

C core in C++

1. In its guts C++ still shares many things with C
2. Compatibility with C was a design goal for the C++ committee
3. The classification of types is similar to the one you may find in C...
4. ...with the exception of classes (and that's where everything began)
5. C++ permits function overloading, while C does not
6. C++ introduces the use of namespaces to avoid name cluttering

Best practices

1. Use include guards to prevent violations of the **ODR**
2. Avoid pre-processor macros and use **const** whenever possible
3. Prefer **std**::**vector** over built-in arrays

Table of contents - Appendices


Meyers, S.
Effective C++: 55 Specific Ways to Improve Your Programs and Designs, 3rd ed.
Addison-Wesley Professional, 2005.


Sutter, H., and Alexandrescu, A.
C++ Coding Standards: 101 Rules, Guidelines, and Best Practices, 1 ed.
Addison-Wesley Professional, Nov. 2004.

Table of contents - Appendices

Preprocessing directives (§16)

A preprocessing directive consists of a sequence of preprocessing tokens

- The first token in the sequence is a #
- The last token is the first newline character that follows #

The most common uses of a preprocessing directive are:

- Conditional inclusion
- Source file inclusion
- Macro replacement
- Pragma directives

Preprocessing tokens within a directive are not subject to macro expansion.

Conditional inclusion (§16.1)

Expression controlling conditional inclusion shall be integral constant expressions, with the exception of:

```
#if defined MACRONAME
<code>
#endif
```

and:

```
#if defined (MACRONAME)
<code>
#endif
```

The unary operator expression `defined` evaluates to:

- 1 if the identifier is currently defined as a macro name
- 0 if it is not

Conditional inclusion (§16.1)

Preprocessing directives of the form:

```
#if first_constant_expression
<first_branch>
#elif second_constant_expression
<second_branch>
#else
<default_branch>
#endif
```

check the corresponding constant expressions, and preprocess the first that evaluates to true. The following shortened forms are defined:

```
#ifdef id // #if defined id
#ifndef id // #if !defined id
```

Source file inclusion (§16.2)

A preprocessing directive of the form:

```
#include <header_file>
```

searches a sequence of implementation-defined places for a header. How the places are specified or the header identified is implementation-defined.

A preprocessing directive of the form:

```
#include "header_file"
```

searches for the header in an implementation-defined manner. If this search fails the directive reverts to the previous case.

Both directives cause the replacement of that directive by the entire contents of the source file identified by the specified sequence.

Macro replacement (§16.3)

A preprocessing directive of the form:

```
#define identifier replacement
```

defines an object-like macro that causes the replacement of each subsequent instance of the macro name.

A preprocessing directive of the form:

```
#define macro_name(identifier) replacement
```

defines a function-like macro with parameters, whose use is similar syntactically to a function call.

Within the sequence of preprocessing tokens making up an invocation of a function-like macro, new-line is considered a normal white-space character.

Macro replacement (§16.3)

The # operator (Stringification)

- shall be followed by a parameter
- is replaced by a single character string literal
- the order of evaluation of # and ## operators is unspecified.

The ## operator (Concatenation)

- concatenates two pre-processing tokens
- the order of evaluation of ## operators is unspecified

Argument substitution

- takes place after the arguments of a function-like macro have been identified
- a parameter not preceded by # or ## is replaced by the corresponding argument
- if a macro is given as an argument, it is expanded before being substituted

Q: What are the stages of the following macro expansion?

```
#define dhash ####
#define mkstr(a) # a
#define in_between(a) mkstr(a)
#define join(c, d) in_between(c dhash d)

char p[] = join(x, y);
```

Predefined macro names (§16.8)

The following macro names shall be defined by the implementation:

```
__cplusplus
// defined for CXX translation units
__DATE__
// date of translation of the source file
__TIME__
// time of translation of the source file
__FILE__
// presumed name of the source file
```

If any of the pre-defined macro names is re-defined or un-defined, this triggers an undefined behavior.

The **sizeof** operator (§5.3.3)

The **sizeof** operator: :

```
float a[10];

int nelems = sizeof( a ) / sizeof( a[0] );
```

yields the number of bytes in the object representation of its operand.

1. **sizeof** shall not be applied to a function or incomplete type
2. **sizeof**(**char**), **sizeof**(**signed char**) and **sizeof**(**unsigned char**) are 1
3. For other fundamental types, the result is implementation defined
4. For references or a reference types, the result is the size of the referenced type

The **new** (§5.3.4) and **delete** (§5.3.5) expressions

```
int *  pi  = new int;
delete pi;

// Parentheses needed for compound types
int (*)() pfa = new ( int (*[10])() );
delete[]  pfa;

// The return type is int (*)[10]
int (*)[10] pia = new int[20][10];
delete[]    pia;
```

1. The new-expression attempts to create an object of a given type
2. The delete-expression destroys an object created by a new-expression
3. Entities created this way have dynamic storage duration

The **static** specifier (§7.1.1)

The **static** specifier applied to functions implies internal linkage:

```
// internal linkage
static char* f();
char* f() {
 /* ... */
}
```

while applied to variables it implies static storage duration:

```
int getID() {
  static int count = 0;
  count++;
  return count;
}
```

The **inline** specifier (§7.1.2)

A function declaration with an **inline** specifier declares an inline function.

The specifier indicates to the implementation that inline substitution of the function body at the point of call is to be preferred to the usual function call mechanism:

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

An implementation is not required to perform this inline substitution at the point of call.

The **typedef** specifier (§7.1.3)

The keyword **typedef** declares identifiers that can be used later for naming fundamental or compound types.

A name declared with the **typedef** specifier becomes a typedef-name.

A typedef-name is a synonym for another type. A typedef-name does not introduce a new type the way a class or enum declaration does.

```
typedef int value_type;
typedef int (*handler)(value_type);

// fpointer has type int (*)(int)
handler fpointer;
```

const and volatile qualifiers (§7.1.6)

There are two cv-qualifiers, **const** and **volatile**.

A pointer or reference to a cv-qualified type need not actually point or refer to a cv-qualified object, but it is treated as if it does.

A const-qualified access path cannot be used to modify an object:

```
int         a   = 10;
int *       pi  = &a;
const int * cpi = &a;
*pi  = 20; // OK
*cpi = 30; // ERROR
```

The keyword **volatile** is a hint to the implementation that the value of an object might be changed at any time.