Các kiểu dữ liệu và khai báo

Các thuật ngữ

Declaration
type
object
value
variable
implementation-defined
unspecified
undefined

Kiểu int có kích thước là bao nhiêu?

Char a= 1234 thì giá trị a lưu là bao nhiêu

Mảng a[100], gán a[101] = 1?

Kiểu dữ liệu

A Boolean type : bool {true - !0/false = 0} {convert to int}
Character types : **char, unsigned char, signed char,**
wchar_t, char16_t, char32_t
Integer types : int, long, long long, usigned int, ...
floating-point types: float, double, long double
void:
Size_t
Alignment

Pointer
Array
referrence

Struct
Class
enum

Khai báo

Declaration/ defination

Optional prefix specifiers (e.g., static or virtual)	A base type (e.g., vector<double> or const int)	A declarator optionally including a name (e.g., p[7], n, or *(*)[])	Optional suffix function specifiers (e.g., const or noexcept)	An optional initializer or function body (e.g., ={7,5,3} or {return x;})

```
const char* kings[] = { "Antigonus", "Seleucus", "Ptolemy" };

int* p, y;          // int* p; int y;      NOT int* y;
int x, *q;          // int x; int* q;
int v[10], *pv;     // int v[10]; int* pv;
```

Khởi tạo

```
X a1 {v};
X a2 = {v};
X a3 = v;
X a4(v);
```

```
int x4 {};          // x4 becomes 0
double d4 {};       // d4 becomes 0.0
char* p {};         // p becomes nullptr
vector<int> v4{};   // v4 becomes the empty vector
string s4 {};       // s4 becomes ""
```

```
void f(double val, int val2)
{
    int x2 = val;       // if val==7.9, x2 becomes 7
    char c2 = val2;     // if val2==1025, c2 becomes 1

    int x3 {val};       // error: possible truncation
    char c3 {val2};     // error: possible narrowing
```

Kiểu dữ liệu suy luận

auto : sinh ra nhờ kiểu của object được gán

decltype() : sinh ra nhờ

```
auto v1 = 12345;       //

auto v1 {12345};       // v1 is a list of int
```

example:

```
template<class T> void f1(vector<T>& arg)
{
    for (vector<T>::iterator p = arg.begin(); p!=arg.end(); ++p)
        *p = 7;

    for (auto p = arg.begin(); p!=arg.end(); ++p)
        *p = 7;
}
```

Type Aliases

or example:

```
using Pchar = char*;          // pointer to character
using PF = int(*)(double);    // pointer to function taking a double and returning an int
```

```
typedef int int32_t;          // equivalent to "using int32_t = int;"
typedef short int16_t;        // equivalent to "using int16_t = short;"
typedef void(*PtoF)(int);     // equivalent to "using PtoF = void(*)(int);"
```

Scope and life time

Local scope : local name – function and lamda {}
Class scope : thành viên lớp
Namespace scope :
Global scope :
Statement scope :
Function scope :

Automatic : , automatic objects are allocated on the stack
Static :
Free store:
Temporary objects : tính giá trị biểu thức.
Thread-local :

Phases of translation

Phase 1

1) The individual bytes of the source code file are mapped (in implementation-defined manner) to the characters of the *basic source character set*. In particular, OS-dependent end-of-line indicators are replaced by newline characters. The *basic source character set* consists of 96 characters:

   a) 5 whitespace characters (space, horizontal tab, vertical tab, form feed, new-line)

   b) 10 digit characters from `'0'` to `'9'`

   c) 52 letters from `'a'` to `'z'` and from `'A'` to `'Z'`

   d) 29 punctuation characters: `_ { } [ ] # ( ) < > % : ; . ? * + - / ^ & | ~ ! = , \ " '`

2) Any source file character that cannot be mapped to a character in the basic source character set is replaced by its universal character name (escaped with \u or \U) or by some implementation-defined form that is handled equivalently.

3) Trigraph sequences are replaced by corresponding single-character representations. (until C++17)

Phase 2

1) Whenever backslash appears at the end of a line (immediately followed by the newline character), both backslash and newline are deleted, combining two physical source lines into one logical source line. This is a single-pass operation; a line ending in two backslashes followed by an empty line does not combine three lines into one. If a universal character name (`\uXXXX`) is formed in this phase, the behavior is undefined.

2) If a non-empty source file does not end with a newline character after this step (whether it had no newline originally, or it ended with a backslash), the behavior is undefined (until C++11) a terminating newline character is added (since C++11).

Phase 3

1) The source file is decomposed into comments, sequences of whitespace characters (space, horizontal tab, new-line, vertical tab, and form-feed), and *preprocessing tokens*, which are the following:

   a) header names such as `<iostream>` or `"myfile.h"`

   b) identifiers

   c) preprocessing numbers

   d) character and string literals , including user-defined (since C++11)

   e) operators and punctuators (including alternative tokens), such as `+` , `<<=` , `new` , `<%` , `##` , or `and`

   f) individual non-whitespace characters that do not fit in any other category

2) Any transformations performed during phases 1 and 2 between the initial and the final double quote of any raw string literal are reverted. (since C++11)

3) Each comment is replaced by one space character.

Phase 4

1) The preprocessor is executed.

2) Each file introduced with the #include directive goes through phases 1 through 4, recursively.

3) At the end of this phase, all preprocessor directives are removed from the source.

3) At the end of this phase, all preprocessor directives are removed from the source.

Phase 5

1) All characters in character literals and string literals are converted from the source character set to the *execution character set* (which may be a multibyte character set such as UTF-8, as long as the 96 characters of the *basic source character set* listed in phase 1 have single-byte representations).
2) Escape sequences and universal character names in character literals and non-raw string literals are expanded and converted to the *execution character set*. If the character specified by a universal character name isn't a member of the execution character set, the result is implementation-defined, but is guaranteed not to be a null (wide) character.

Note: the conversion performed at this stage can be controlled by command line options in some implementations: gcc and clang use `-finput-charset` to specify the encoding of the source character set, `-fexec-charset` and `-fwide-exec-charset` to specify the encodings of the execution character set in the string and character literals that don't have an encoding prefix (since C++11), while Visual Studio 2015 Update 2 and later uses `/source-charset` and `/execution-charset` to specify the source character set and execution character set respectively.

Phase 6

Adjacent string literals are concatenated.

Phase 7

Compilation takes place: each preprocessing token is converted to a token. The tokens are syntactically and semantically analyzed and translated as a translation unit.

Phase 8

Each translation unit is examined to produce a list of required template instantiations, including the ones requested by explicit instantiations. The definitions of the templates are located, and the required instantiations are performed to produce *instantiation units*.

Phase 9

Translation units, instantiation units, and library components needed to satisfy external references are collected into a program image which contains information needed for execution in its execution environment.