Modern C++ - Declarations

10. Declarations

Declarations and definitions

Declaration - tells the compiler what should it think about X.

- For variables: tells the compiler that the variable with a certain type exists (defined somewhere).
- For functions: tells the signature (return type and parameter list). The compiler knows whether conversion is required on parameter passing.
- For types: tells the compiler, that a type (struct, enum, etc. exists).

Definition - tells the compiler what is X in details.

- For variables: tells the type, the life and the scope. Allocate memory for that object.
- For functions: describe parameter list, return type and the algorithm to execute.
- For types: describe the data structure, and members.

The generic form of definitin/declaration is: *storage-class type-name declarator-list*

where *declarator-list* is: *declarator , declarator ...*

storage-class is: *auto*, *register*, *static*, *extern*, *typedef*

Do not use *auto* and *register*. Auto has a different meaning in C++11 and register optimizations are done by modern compilers automatically.

The form of *declarator* is:

- function: **decl (param-list)**
- pointer: * **decl**
- reference **&decl**
- array: **decl [n]**
- otherwise: **id**

Definition

Every C++ element should be defined exactly ones, this rule is called as the *One definition rule* (ODR).

Functions should be defined outside of other functions; i.e. there are no inner functions. Variables can be defined outside (*global* variables) or inside (*local* variables) of functions.

In C++ the place of a variable definition can be in anywhere, even between two executable statements:

```
 1 void func()
 2 {
 3   printf("Hello");
 4   int i = 0;  // definition and initialization
 5   while ( i < 10 )
 6   {
 7     //...
 8     ++i;
 9   }
10 }
```

Variable and function definitions

```
 1 int i;            // integer variable
 2 int *pi;          // pointer to integer
 3 int &ri = i;      // reference to integer, must be initialized
 4 int t[10];        // array of 10 integers
 5 int func(){...}        // function without parameters, returning int
 6 int func(void){...}    // also means no parameter, C reverse compatible
 7 int func(int i, double d){...} // function, i and d are formal parameters
 8 void func(){...}  // function without parameters, no return type
 9
10 //...
11 char *getenv(const char *var){...}
```

These definitions can be used recursively

```
1 int **ppi;       // pointer to a pointer to int
2 int tt[10][20];  // array of 10 arrays of 20 integers (200 int elements)
3 int *func(int){...}  // int parameter function returning pointer to int
```

In case of disambiguity, usual precedence rules decide the meaning. We can use parenthesis to overrule precedence.

```
1 int  *ptr_arr[10];   // array of 10 pointers to integers
2 int (*ptr_to)[10];   // pointer to an array of 10 integers
```

But there are a few exeptions:

- No array of functions (but there is array of *function pointers*).
- No functions returning functions (but can return with *function pointer*).

Pointers to functions should define full signature.

```
1 double (*funcptr)(double); // pointer to double f(double) func, like sin
2 //...
3 extern double sin(double); // declaration of sin, usually in <math.h>
4 funcptr = sin;             // name of function is pointer to function value
```

Sometimes **typedef** is used to simplify definitions/declarations.

```
1 typedef double length_t;   // length_t is synonym of double
2 length_t f(length_t){...}  // f has a double parameter and returns double
3 //...
4 typedef double (*trigfp_t)(double);// trigfp_t is pointer to function
5                                    // signature is part of the type
6 trigfp_t inverse(trigfp_t){...}    // function returning pointer to function
7 int  trigfp_t[10];         // array of 10 pointer to double(double) function
```

Arrays with automatic life (local, non-static arrays) can be *variadic*, i.e. the size of an array can be a value of a variable.

```
1 static int t1[10];  // dimension must be known compile time
2
3 void f()
4 {
5   static int t2[10];  // dimension must be known compile time
6   int n;
7   // read n
8   int t3[n];          // ok
9 }
```

Declaration

Every C++ elements should be declared before the first usage. Multiply declarations are allowed.

```
extern int i;   // integer variable is defined somewhere else
extern int *pi; // pointer to integer is defined somewhere else
extern int t[10];      // array of 10 integers
extern int t[];        // array of ? integers
extern int tt[10][20]; // array of 10x20 integers
extern int tt[][20];   // ?x20 integers, only leftmost dim can be omitted
extern int func();     // function without parameters, returning int
int func();            // same, compiler understands: this is declaration
int func(void);        // also means no parameter, C reverse compatible
static int func();     // func has no external linkage, parameters unknown
//...
/* extern */ char *getenv();
```

Initialization

Variables can be initialized when defined. Actually, the best strategy is to delay definitions until we use and initialize the variable.

References and constant variables *should* be initialized!

```
int i = 1;
int &ir = i;             // reference must be initialized
const double pi = 3.14;  // const must be initialized

extern double sin(double);  // declaration
double (*funcptr)(double) = sin;

int arr1[10] = {0,1,2,3,4,5,6,7,8,9};  // ok, but hard to maintain
int arr2[10] = {0,1,2,3,4,5,6,7,8};    // arr2[9] == 0
int arr3[]   = {0,1,2,3,4,5,6,7,8};    // int arr3[9]

char str1[] = {'H','e','l','l','o','\0'}; // char str1[6]
char str2[] = "Hello";                    // char str2[6]
```

Forward declarations

There are cases, when two type definitions should refer to each other, therefore we can not serialize their declarations. In these situations (and some other cases) we shoudl use *forward declarations*.

```
 1 struct Current;
 2
 3 struct Other
 4 {
 5     Current *cp;
 6 };
 7
 8 struct Current
 9 {
10     Other *op;
11 };
```

External contants

Constants are local to the translation unit by default. However, we can define and declare cross-translation unit constants.

```
1 const int bufsize = 1024;             // non-extern const
2 extern const int global_const;        // extern const declaration
3 extren const int global_const = 80;   // extern const definition
```

Extern constants should be defined in a single translation unit, here will be the memory allocated for the constant. This is the place when we initialize it (line 2). In the other translation units we should declare the constant (line 3).

Pitfalls

There are different styles to declare pointers regarding where we put the start declarator. However, sometimes the syntax can be misleading.

```
1 int   i,  j;   // i and j are integers
2 int* ip,  jp;  // ip is pointer, but jp is integer
3 int *ip, *jp;  // ok, both ip and jp are pointers
```

In the second case, only **ip** will be declared to pointer, the **jp** variable will be integer.