typedef

From Wikipedia, the free encyclopedia

typedef is a reserved keyword in the C and C++ programming languages. It is used to create an alias name for another data type.[1] As such, it is often used to simplify the syntax of declaring complex data structures consisting of struct and union types, but is just as common in providing specific descriptive type names for integer data types of varying lengths. The C standard library and POSIX reserve the suffix '_t', for example as in size_t and time_t.

Contents

Syntax

The syntax for a creating a typedef is: **typedef** *typedeclaration*;[2]

Some examples:

```
typedef int Length;
```

creates Length as a synonym for int.

```
typedef int (*PFI)(char *, char *);
```

creates PFI as a synonym for a pointer to a function of two char * arguments that returns an int.

Usage examples

Indicating the meaning of a variable definition

A typedef declaration may be used to indicate the meaning of a variable within the programming context, e.g., it may be include the expression of a unit of measurement or counts. The generic declarations in the following code,

```
int current_speed ;
int high_score ;


void congratulate(int your_score) {
    if (your_score > high_score)
        ...
```

may be expressed by declaring context specific types:

```
typedef int km_per_hour ;
typedef int points ;

km_per_hour current_speed ;  //"km_per_hour" is synonymous with "int" here,
points high_score ;          //and thus, the compiler treats our new variables as integers.


void congratulate(points your_score) {
    if (your_score > high_score)
        ...
```

Both sections of code execute identically. However, the use of typedef declarations in the second code block makes it clear that the two variables, while represented by the same data type *int*, represent different or incompatible data. The definition in *congratulate()* of *your_score* indicates to the programmer that *current_speed* (or any other variable not declared as a *points*) should not be passed as an argument. This would not be as apparent if both were declared as variables of *int* datatype. However, the indication is *for the programmer only*; the C/C++ compiler considers both variables to be of type *int* and does not flag type mismatch warnings or errors for "wrong" argument types for *congratulate(points your_score)* in the code snippet below:

```
void foo() {
    km_per_hour km100 = 100;
    congratulate(km100);
```

Although the compiler considers *km_per_hour* to be equivalent to *int* in the above code, the two cannot be used interchangeably when the type is changed via a prefix of *unsigned*, *signed*, or *long*.

```
void foo() {
    unsigned int a;         // Okay
    unsigned km_per_hour b; // Compiler complains
    long int c;             // Okay
    long km_per_hour d;     // Compiler complains
```

Simplifying a definition or declaration

A typedef may be used to simplify the declaration of a compound type (struct, union) or pointer type.[3] For example, in the following snippet:

```
struct MyStruct {
    int data1;
    char data2;
};
```

the data type *struct MyStruct* is defined. To declare a variable of this type in C, the *struct* key word is required (in C++, it may be omitted):

```
struct MyStruct a;
```

A typedef may be used to eliminate the need for the keyword *struct* in C. For example, the following declaration

```
typedef struct MyStruct newtype;
```

may be used to define an object of type *newtype*:

```
newtype a;
```

The structure declaration and definition may instead be combined into a single statement:

```
typedef struct MyStruct {
    int data1;
    char data2;
} newtype;
```

Or it may be used as follows:

```
typedef struct {
    int data1;
    char data2;
} newtype;
```

In C++, in contrast to C, the keywords *struct*, *class*, and *enum* are optional in variable declarations that are separate from the definitions, as long as there is no ambiguity to another identifier:

```
struct MyStruct x;       // This is legal
MyStruct y;              // This is also legal
```

As such, *MyStruct* can be used wherever *newtype* can be used. However, the reverse is not true; for instance, the constructor methods for *MyStruct* cannot be named *newtype*.

A notorious example where even C++ needs the *struct* keyword is POSIX' stat system call that uses a *struct* of the same name in its arguments:

```
int stat(const char *filename, struct stat *buf)
{
    // codes are here
}
```

Here both C as well as C++ need the *struct* keyword in the parameter definition.

Using typedef with pointers

We can use the typedef to define a new pointer type. For example,

```
typedef int *intptr;    // type name: intptr
                        // new type: int*

intptr ptr;             // same as: int *ptr
```

Above, *intptr* is a new alias with the pointer type *int**. The definition, *intptr ptr;*, defines a variable *ptr* with the type *int**. So, *ptr* is a pointer which can point to a memory with *int* type.

Using *typedef* to define a new pointer type may sometimes lead to confusion. For example:

```
typedef int *intptr;

intptr cliff, allen;        // both cliff and allen are int* type
intptr cliff2, *allen2;     // cliff2 is int* type, but allen2 is int** type
                             // same as: intptr cliff2;
                             //          intptr *allen2;
```

Above, *intptr cliff, allen;* means defining 2 variables with *int** type for both. This is because a type defined by *typedef* is a type, not an expansion. In other words, *intptr*, which is the *int** type, decorates both cliff and allen. For *intptr cliff2, *allen2;*, the *intptr* type decorates the *cliff2* and **allen2*. So, *intptr cliff2, *allen2;* is equivalent to 2 separate definitions, *intptr cliff2;* and *intptr *allen2*. *intptr *allen2* means that *allen2* is a pointer pointing to a memory with *int** type. Shortly, *allen2* has the type, *int***.

Using typedef with structur e pointers

Typedefs can also simplify definitions or declarations for structure pointer types. Consider this:

```
struct Node {
    int data;
    struct Node *nextptr;
};
```

Using typedef, the above code can be rewritten like this:

```
typedef struct Node Node;
struct Node {
    int data;
    Node *nextptr;
};
```

In C, one can declare multiple variables of the same type in a single statement, even mixing structure with pointer or non-pointers. However, one would need to prefix an asterisk to each variable to designate it as a pointer. In the following, a programmer might assume that *errptr* was indeed a *Node **, but a typographical error means that *errptr* is a *Node*. This can lead to subtle syntax errors.

```
struct Node *startptr, *endptr, *curptr, *prevptr, errptr, *refptr;
```

By defining the typedef *Node **, it is assured that all variables are structure pointer types, or say, that each variable is a pointer type pointing to a structure type.

```
typedef struct Node* NodePtr;
...
NodePtr startptr, endptr, curptr, prevptr, errptr, refptr;
```

Using typedef with function pointers

Consider the following code, which does not use a typedef:

```
int do_math(float arg1, int arg2) {
    return arg2;
}

int call_a_func(int (*call_this)(float, int)) {
    int output = call_this(5.5, 7);
    return output;
}

int final_result = call_a_func(&do_math);
```

This code can be rewritten with a typedef as follows:

```
typedef int (*MathFunc)(float, int);

int do_math(float arg1, int arg2) {
    return arg2;
}

int call_a_func(MathFunc call_this) {
    int output = call_this(5.5, 7);
    return output;
}

int final_result = call_a_func(&do_math);
```

Here, *MathFunc* is the new alias for the type. A *MathFunc* is a pointer to a function that returns an integer and takes as arguments a float followed by an integer.

When a function returns a function pointer, it can be even more confusing without typedef. The following is the function prototype of *signal(3)* from FreeBSD:

```
void (*signal(int sig, void (*func)(int)))(int);
```

The function declaration above is cryptic as it does not clearly show what the function accepts as arguments, or the type that it returns. A novice programmer may even assume that the function accepts a single *int* as its argument and returns nothing, but in reality it also needs a function pointer and returns another function pointer. It can be written more cleanly:

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int sig, sighandler_t func);
```

Using typedef with arrays

A typedef can also be used to simplify the definition of array types. For example,

```
typedef char arrType[6];    // type name: arrType
                            // new type: char[6]

arrType arr={1,2,3,4,5,6};  // same as: char arr[6]={1,2,3,4,5,6}

arrType *pArr;              // same as: char (*pArr)[6];
```

Here, arrType is the new alias for the "char[6]" type, which is an array type with 6 elements. For *arrType *pArr;*, pArr is a pointer pointing to the memory of the "char[6]" type.

Using typedef with type casts

A typedef is created using type *definition* syntax but can be used as if it were created using type *cast* syntax. (Type casting changes a data type.) For instance, in each line after the first line of:

```
typedef int (*funcptr)(double);         // pointer to function taking a double returning int
funcptr x = (funcptr) NULL;             // C or C++
funcptr y = funcptr(NULL);              // C++ only
funcptr z = static_cast<funcptr>(NULL); // C++ only
```

funcptr is used on the left-hand side to declare a variable and is used on the right-hand side to cast a value. Thus, the typedef can be used by programmers who do not wish to figure out how to convert definition syntax to type cast syntax.

Note that, without the typedef, it is generally not possible to use definition syntax and cast syntax interchangeably. For example:

```
void *p = NULL;
int (*x)(double)  = (int (*)(double)) p; // This is legal
int (*)(double) y = (int (*)(double)) p; // Left-hand side is not legal
int (*z)(double)  = (int (*p)(double));  // Right-hand side is not legal
```

Usage concerns

Some people are opposed to the extensive use of typedefs. Most arguments center on the idea that typedefs simply hide the actual data type of a variable. For example, Greg Kroah-Hartman, a Linux kernel hacker and documenter, discourages their use for anything except function prototype declarations. He argues that this practice not only unnecessarily obfuscates code, it can also cause programmers to accidentally misuse large structures thinking them to be simple types.[4]

Others argue that the use of typedefs can make code easier to maintain. K&R states that there are two reasons for using a typedef. First, it provides a means to make a program more portable. Instead of having to change a type everywhere it appears throughout the program's source files, only a single typedef statement needs to be changed. Second, a typedef can make a complex definition or declaration easier to understand.

Usage in C++

In C++ type names can be very complicated and typedef provides a mechanism to assign a simple name to the type. Consider:

```
std::vector<std::pair<std::string, int> > values;
for (std::vector<std::pair<std::string, int> >::const_iterator i = values.begin(); i != values.end(); ++i)
{
   std::pair<std::string, int> const & t = *i;
   // do something
}
```

and

```
typedef std::pair<std::string, int> value_t;
typedef std::vector<value_t> values_t;

values_t values;
for (values_t::const_iterator i = values.begin(); i != values.end(); ++i)
{
   value_t const & t = *i;
   // do something
}
```

Use with templates

C++03 does not provide templated typedefs. For instance, to have *stringpair<T>* represent *std::pair<std::string, T>* for every type *T* one *cannot* use:

```
template<typename T>
typedef std::pair<std::string, T> stringpair<T>; // Doesn't work
```

However, if one is willing to accept *stringpair<T>::type* in lieu of *stringpair<T>* then it is possible to achieve the desired result via a typedef within an otherwise unused templated *class* or *struct*:

```
template<typename T>
class stringpair
{
private:
    // Prevent instantiation of stringpair<T>
    stringpair();
public:
    // Make stringpair<T>::type represent std::pair<std::string, T>
    typedef std::pair<std::string, T> type;
};

// Declare a variable of type std::pair<std::string, int>
stringpair<int>::type my_pair_of_string_and_int;
```

In C++11, templated typedefs are added with the following syntax, which requires the *using* keyword rather than the *typedef* keyword. (See template aliases.)

```
template <typename T>
using stringpair = std::pair<std::string, T>;

// Declare a variable of type std::pair<std::string, int>
stringpair<int> my_pair_of_string_and_int;
```

Other languages

In many statically typed functional languages, like Haskell, Miranda, OCaml, etc., one can define *type synonyms*, which are the same as typedefs in C. An example in Haskell:

```
type PairOfInts = (Int, Int)
```

This example has defined a type synonym *PairOfInts* as an integer type.

In Seed7 the definition of a constant type is used to introduce a synonym for a type:

```
const type: myVector is array integer;
```

In Swift, typedef is called *typealias*:

```
typealias PairOfInts = (Int, Int)
```

C# also contains a feature which is similar to the typedef of C.[5]

```
using newType = global::System.Runtime.Interop.Marshal;
using otherType = Enums.MyEnumType;
using StringListMap = System.Collections.Generic.Dictionary<string, System.Collections.Generic.List<string>>;
```

In D the keyword *alias*[6] allows to create type or partial type synonyms.

```
struct Foo(T){}
alias FooInt = Foo!int;
alias Fun = int delegate(int);
```

See also

- Abstract data type
- C syntax

References

1. Kernighan, Brain W.; Ritchie, Dennis M. (1988). *The C Programming Language* (https://www.pearsonhighered.com/program/Kernighan-C-Programming-Language-2nd-Edition/PGM54487.html) (2nd ed.).

Englewood Cliffs, New Jersey.: Prentice Hall. p. 147. ISBN 0-13-110362-8. Retrieved 18 June 2016. "C provides a facility called typedef for creating new data type names. … It must be emphasized that a typedef declaration does not create a new type in any sense; it merely adds a new name for some existing type."

2. "typedef specifier" (http://en.cppreference.com/w/cpp/keyword/typedef). *cppreference.com*. Retrieved 18 June 2016.
3. Deitel, Paul J.; Deitel, H. M. (2007). *C how to program* (http://www.pearsonhighered.com/pearsonhigheredus/educator/product/products_detail.page?isbn=0132404168) (5th ed.). Upper Saddle River, N.J.: Pearson Prentice Hall. ISBN 9780132404167. Retrieved 12 September 2012. "Names for structure types are often defined with *typedef* to create shorter type names."
4. Kroah-Hartman, Greg (2002-07-01). "Proper Linux Kernel Coding Style" (http://www.linuxjournal.com/article/5780). *Linux Journal*. Retrieved 2007-09-23. "Using a typedef only hides the real type of a variable."
5. http://msdn.microsoft.com/en-us/library/aa664765(VS.71).aspx
6. "Declarations - D Programming Language" (https://dlang.org/spec/declaration.html#AliasDeclaration). *dlang.org*. Retrieved 2017-05-28.

Retrieved from "https://en.wikipedia.org/w/index.php?title=Typedef&oldid=800401943"