Chapter 3: Data Types

Section 3.1: Interpreting Declarations

A distinctive syntactic peculiarity of C is that declarations mirror the use of the declared object as it would be in a normal expression.

The following set of operators with identical precedence and associativity are reused in declarators, namely:

- the unary `*` "dereference" operator which denotes a pointer;
- the binary `[]` "array subscription" operator which denotes an array;
- the (1+n)-ary `()` "function call" operator which denotes a function;
- the `()` grouping parentheses which override the precedence and associativity of the rest of the listed operators.

The above three operators have the following precedence and associativity:

Operator	Relative Precedence	Associativity
`[]` (array subscription)	1	Left-to-right
`()` (function call)	1	Left-to-right
`*` (dereference)	2	Right-to-left

When interpreting declarations, one has to start from the identifier outwards and apply the adjacent operators in the correct order as per the above table. Each application of an operator can be substituted with the following English words:

Expression	Interpretation
`thing[X]`	an array of size `X` of...
`thing(t1, t2, t3)`	a function taking `t1`, `t2`, `t3` and returning...
`*thing`	a pointer to...

It follows that the beginning of the English interpretation will always start with the identifier and will end with the type that stands on the left-hand side of the declaration.

Examples

```
char *names[20];
```

`[]` takes precedence over `*`, so the interpretation is: `names` is an array of size 20 of a pointer to `char`.

```
char (*place)[10];
```

In case of using parentheses to override the precedence, the `*` is applied first: `place` is a pointer to an array of size 10 of `char`.

```
int fn(long, short);
```

There is no precedence to worry about here: `fn` is a function taking `long`, `short` and returning `int`.

```
int *fn(void);
```

The `()` is applied first: `fn` is a function taking `void` and returning a pointer to `int`.

```
int (*fp)(void);
```

Overriding the precedence of `()`: `fp` is a pointer to a function taking `void` and returning `int`.

```
int arr[5][8];
```

Multidimensional arrays are not an exception to the rule; the `[]` operators are applied in left-to-right order according to the associativity in the table: `arr` is an array of size 5 of an array of size 8 of `int`.

```
int **ptr;
```

The two dereference operators have equal precedence, so the associativity takes effect. The operators are applied in right-to-left order: `ptr` is a pointer to a pointer to an `int`.

Multiple Declarations

The comma can be used as a separator (*not* acting like the comma operator) in order to delimit multiple declarations within a single statement. The following statement contains five declarations:

```
int fn(void), *ptr, (*fp)(int), arr[10][20], num;
```

The declared objects in the above example are:

- `fn`: a function taking `void` and returning `int`;
- `ptr`: a pointer to an `int`;
- `fp`: a pointer to a function taking `int` and returning `int`;
- `arr`: an array of size 10 of an array of size 20 of `int`;
- `num`: `int`.

Alternative Interpretation

Because declarations mirror use, a declaration can also be interpreted in terms of the operators that could be applied over the object and the final resulting type of that expression. The type that stands on the left-hand side is the final result that is yielded after applying all operators.

```
/*
 * Subscripting "arr" and dereferencing it yields a "char" result.
 * Particularly: *arr[5] is of type "char".
 */
char *arr[20];

/*
 * Calling "fn" yields an "int" result.
 * Particularly: fn('b') is of type "int".
 */
int fn(char);

/*
 * Dereferencing "fp" and then calling it yields an "int" result.
 * Particularly: (*fp)() is of type "int".
 */
int (*fp)(void);

/*
 * Subscripting "strings" twice and dereferencing it yields a "char" result.
 * Particularly: *strings[5][15] is of type "char"
 */
char *strings[10][20];
```

Section 3.2: Fixed Width Integer Types (since C99)

Version ≥ C99

The header `<stdint.h>` provides several fixed-width integer type definitions. These types are *optional* and only provided if the platform has an integer type of the corresponding width, and if the corresponding signed type has a two's complement representation of negative values.

See the remarks section for usage hints of fixed width types.

```
/* commonly used types include */
uint32_t u32 = 32; /* exactly 32-bits wide */

uint8_t u8 = 255;  /* exactly 8-bits wide */

int64_t i64 = -65  /* exactly 64 bit in two's complement representation */
```

Section 3.3: Integer types and constants

Signed integers can be of these types (the `int` after `short`, or `long` is optional):

```
signed char c = 127; /* required to be 1 byte, see remarks for further information. */
signed short int si = 32767; /* required to be at least 16 bits. */
signed int i = 32767; /* required to be at least 16 bits */
signed long int li = 2147483647; /* required to be at least 32 bits. */
```

Version ≥ C99

```
signed long long int li = 2147483647; /* required to be at least 64 bits */
```

Each of these signed integer types has an unsigned version.

```
unsigned int i = 65535;
unsigned short = 2767;
unsigned char = 255;
```

For all types but `char` the `signed` version is assumed if the `signed` or `unsigned` part is omitted. The type `char` constitutes a third character type, different from `signed char` and `unsigned char` and the signedness (or not) depends on the platform.

Different types of integer constants (called *literals* in C jargon) can be written in different bases, and different width, based on their prefix or suffix.

```
/* the following variables are initialized to the same value: */
int d = 42;   /* decimal constant (base10) */
int o = 052;  /* octal constant (base8) */
int x = 0xaf; /* hexadecimal constants (base16) */
int X = 0XAf; /* (letters 'a' through 'f' (case insensitive) represent 10 through 15) */
```

Decimal constants are always `signed`. Hexadecimal constants start with `0x` or `0X` and octal constants start just with a `0`. The latter two are `signed` or `unsigned` depending on whether the value fits into the signed type or not.

```
/* suffixes to describe width and signedness : */
long int i = 0x32; /* no suffix represent int, or long int */
unsigned int ui = 65535u; /* u or U represent unsigned int, or long int */
long int li = 655361; /* l or L represent long int */
```

Without a suffix the constant has the first type that fits its value, that is a decimal constant that is larger than

`INT_MAX` is of type `long` if possible, or `long long` otherwise.

The header file **`<limits.h>`** describes the limits of integers as follows. Their implementation-defined values shall be equal or greater in magnitude (absolute value) to those shown below, with the same sign.

Macro	Type	Value
`CHAR_BIT`	smallest object that is not a bit-field (byte)	8
`SCHAR_MIN`	`signed char`	-127 / -(27 - 1)
`SCHAR_MAX`	`signed char`	+127 / 27 - 1
`UCHAR_MAX`	`unsigned char`	255 / 28 - 1
`CHAR_MIN`	`char`	see below
`CHAR_MAX`	`char`	see below
`SHRT_MIN`	`short int`	-32767 / -(215 - 1)
`SHRT_MAX`	`short int`	+32767 / 215 - 1
`USHRT_MAX`	`unsigned short int`	65535 / 216 - 1
`INT_MIN`	`int`	-32767 / -(215 - 1)
`INT_MAX`	`int`	+32767 / 215 - 1
`UINT_MAX`	`unsigned int`	65535 / 216 - 1
`LONG_MIN`	`long int`	-2147483647 / -(231 - 1)
`LONG_MAX`	`long int`	+2147483647 / 231 - 1
`ULONG_MAX`	`unsigned long int`	4294967295 / 232 - 1

`Version ≥ C99`

Macro	Type	Value
`LLONG_MIN`	`long long int`	-9223372036854775807 / -(263 - 1)
`LLONG_MAX`	`long long int`	+9223372036854775807 / 263 - 1
`ULLONG_MAX`	`unsigned long long int`	18446744073709551615 / 264 - 1

If the value of an object of type `char` sign-extends when used in an expression, the value of `CHAR_MIN` shall be the same as that of `SCHAR_MIN` and the value of `CHAR_MAX` shall be the same as that of `SCHAR_MAX` . If the value of an object of type `char` does not sign-extend when used in an expression, the value of `CHAR_MIN` shall be 0 and the value of `CHAR_MAX` shall be the same as that of `UCHAR_MAX`.

`Version ≥ C99`

The C99 standard added a new header, **`<stdint.h>`**, which contains definitions for fixed width integers. See the fixed width integer example for a more in-depth explanation.

Section 3.4: Floating Point Constants

The C language has three mandatory real floating point types, `float`, `double`, and `long double`.

```
float f = 0.314f;        /* suffix f or F denotes type float */
double d = 0.314;        /* no suffix denotes double */
long double ld = 0.314l; /* suffix l or L denotes long double */

/* the different parts of a floating point definition are optional */
double x = 1.; /* valid, fractional part is optional */
double y = .1; /* valid, whole-number part is optional */

/* they can also defined in scientific notation */
double sd = 1.2e3; /* decimal fraction 1.2 is scaled by 10^3, that is 1200.0 */
```

The header **<float.h>** defines various limits for floating point operations.

Floating point arithmetic is implementation defined. However, most modern platforms (arm, x86, x86_64, MIPS) use IEEE 754 floating point operations.

C also has three optional complex floating point types that are derived from the above.

Section 3.5: String Literals

A string literal in C is a sequence of chars, terminated by a literal zero.

```
char* str = "hello, world"; /* string literal */

/* string literals can be used to initialize arrays */
char a1[] = "abc"; /* a1 is char[4] holding {'a','b','c','\0'} */
char a2[4] = "abc"; /* same as a1 */
char a3[3] = "abc"; /* a1 is char[3] holding {'a','b','c'}, missing the '\0' */
```

String literals are **not modifiable** (and in fact may be placed in read-only memory such as .rodata). Attempting to alter their values results in undefined behaviour.

```
char* s = "foobar";
s[0] = 'F'; /* undefined behaviour */

/* it's good practice to denote string literals as such, by using `const` */
char const* s1 = "foobar";
s1[0] = 'F'; /* compiler error! */
```

Multiple string literals are concatenated at compile time, which means you can write construct like these.

Version < C99

```
/* only two narrow or two wide string literals may be concatenated */
char* s = "Hello, " "World";
```

Version ≥ C99

```
/* since C99, more than two can be concatenated */
/* concatenation is implementation defined */
char* s1 = "Hello" ", " "World";

/* common usages are concatenations of format strings */
char* fmt = "%" PRId16; /* PRId16 macro since C99 */
```

String literals, same as character constants, support different character sets.

```
/* normal string literal, of type char[] */
char* s1 = "abc";

/* wide character string literal, of type wchar_t[] */
wchar_t* s2 = L"abc";
```

Version ≥ C11

```
/* UTF-8 string literal, of type char[] */
char* s3 = u8"abc";

/* 16-bit wide string literal, of type char16_t[] */
char16_t* s4 = u"abc";

/* 32-bit wide string literal, of type char32_t[] */
char32_t* s5 = U"abc";
```