Types

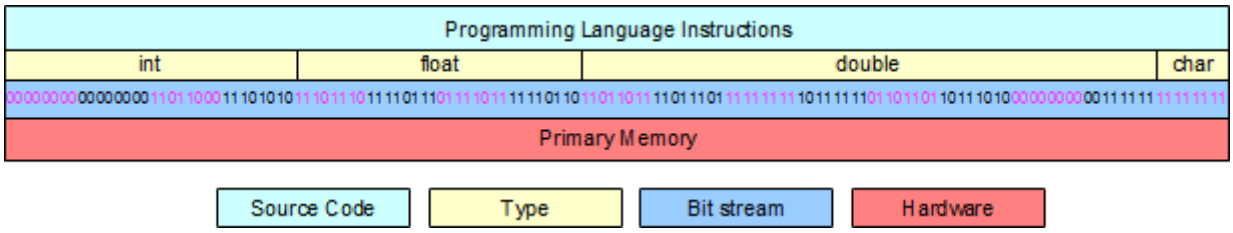
Learning Outcomes

After reading this section, you will be able to:

- Select appropriate types for storing program variables and constants
- Describe the internal representations defined by different types

Introduction

A typed programming language uses a type system to interpret the bit streams in memory. C is a typed programming language. A type is the rule that defines how to store values in memory and which operations are admissible on those values. A type defines the number of bytes available for storing values and hence the range of possible values. We use different types to store different information. The relation between types and raw memory is illustrated in the figure below.



This chapter describes the four most common types in the C language and the ranges of values that these types allow. This chapter concludes by describing how to allocate memory for variables by identifying their contents using a type.

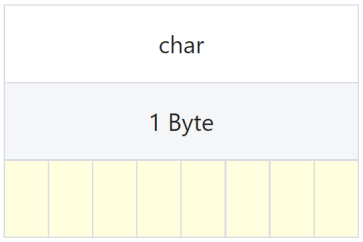
This chapter describes the four most common types in the C language and the ranges of values that these types allow. This chapter concludes by describing how to allocate memory for variables by identifying their contents using a type.

Arithmetic Types

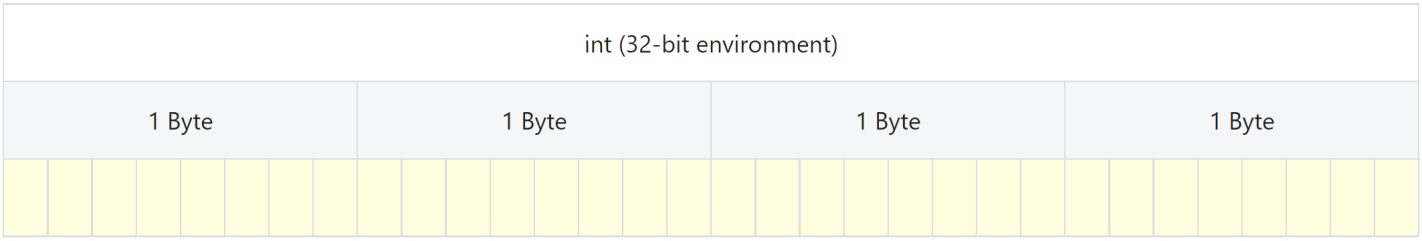
The four most common types of the C language for performing arithmetic calculations are:

- `char`
- `int`
- `float`
- `double`

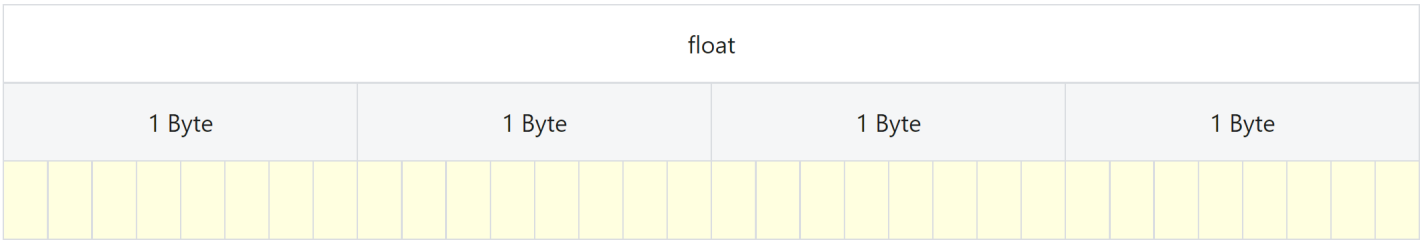
A **`char`** occupies one byte and can store a small integer value, a single character or a single symbol:



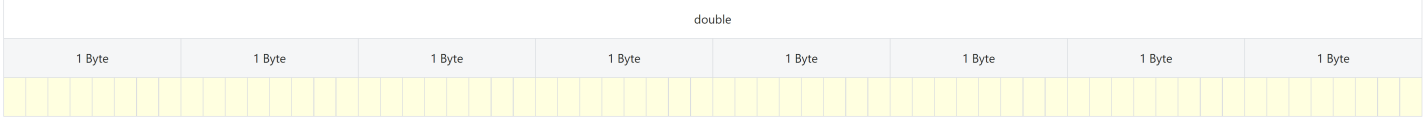
An **`int`** occupies one word and can store an integer value. In a 32-bit environment, an **`int`** occupies 4 bytes:



A **`float`** typically occupies 4 bytes and can store a single-precision, floating-point number:



A **`double`** typically occupies 8 bytes and can store a double-precision, floating-point number:



Size Specifiers

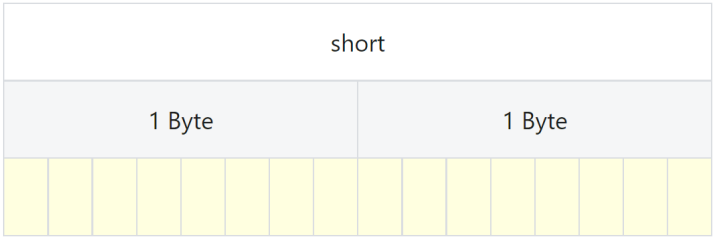
Size specifiers adjust the size of the **`int`** and **`double`** types.

`int` Type Size Specifiers

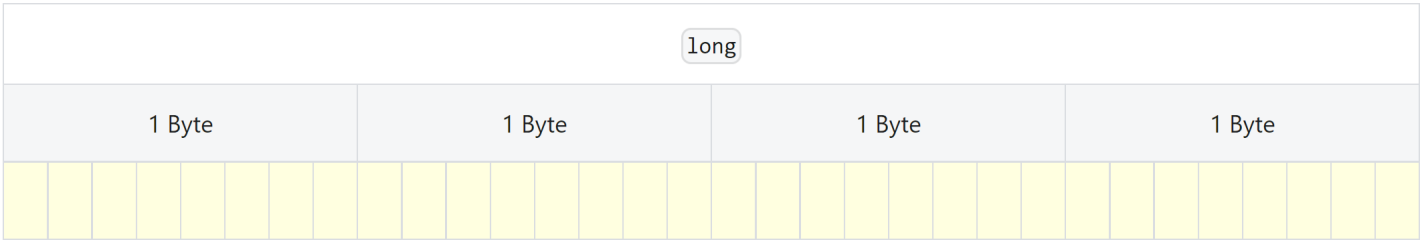
Specifying the size of an int ensures that the type contains a minimum number of bits. The three specifiers are:

- `short`
- `long`
- `long long`

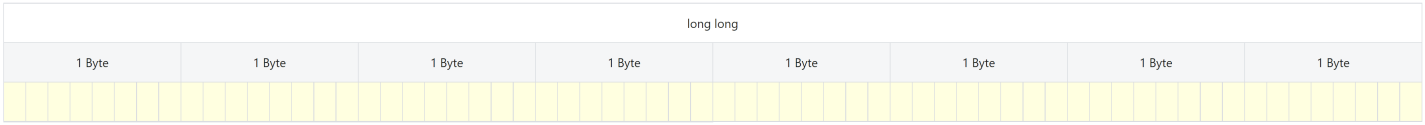
A **`short int`** (or simply, a `short`) contains at least 16 bits:



A **`long int`** (or simply, a `long`) contains at least 32 bits:



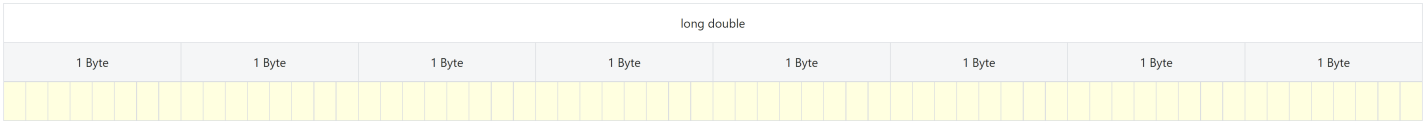
A **`long long int`** (or simply, a `long long`) contains at least 64 bits:



The size of a simple **`int`** is no less than the size of a **`short`**.

`double` Type Size Specifier

The size of a **`long double`** depends on the environment and is typically at least 64 bits:



Specifying the `long double` type only ensures that it contains at least as many bits as a `double`. The C language does not require a `long double` to contain a minimum number of bits.

`const` Qualifier

Any type can hold a constant value. A constant value cannot be changed. To qualify a type as holding a constant value we use the keyword const. A type qualified as `const` is unmodifiable. That is, if a program instruction attempts to modify a `const` qualified type, the compiler will report an error.

Representing Values

Hardware manufacturers distinguish integral types from floating-point types and represent integral data and floating-point data differently.

- integral types: `char` `int`
- floating-point types: `float` `double`

Integral Types

C stores the `char` and `int` data in equivalent binary form. Binary form represents the value stored exactly. To learn how to convert between decimal and binary representation refer to the appendix entitled Data Conversions.

Characters and Symbols

C stores characters and symbols in `char` types. Since characters and symbols have no intrinsic binary representation, the host platform provides the collating sequence for associating each character and symbol with a unique integer value. C stores the integer value from this sequence as the representative of the character or symbol.

The two popular collating sequences are ASCII and EBCDIC. ASCII is more popular. ASCII represents the letter A by the bit pattern 010000012, that is the hexadecimal value 0x41, that is

the decimal value 65. EBCDIC represents the letter A by the bit pattern 110000012, that is the hexadecimal value 0xC1, that is the decimal value 193.

ASCII and EBCDIC are not compatible. The symbol order in ASCII differs from that in EBCDIC. In ASCII, the digits precede the letters, while in EBCDIC, the letters precede the digits. If we sort symbolic information that contains digits and letters, we will obtain different results under each sequence.

Neither ASCII nor EBCDIC contain enough values to represent most of the characters and symbols in the world languages. The Unicode standard, which is compatible with ASCII, provides a much more comprehensive collating system. We use the ASCII collating sequence throughout these notes.

Negative Values (Optional)

There are three schemes for storing negative integers:

- 2's complement notation (most popular)
- 1's complement notation
- sign magnitude notation

All three represent non-negative values identically. Under the 2's complement rule, there is only one representation of 0 and separate addition and subtraction circuits in the ALU are unnecessary.

To obtain the 2's complement of an integer, we

- flip the bits
- add one

For example, we represent the integer -92 by 10100100_2

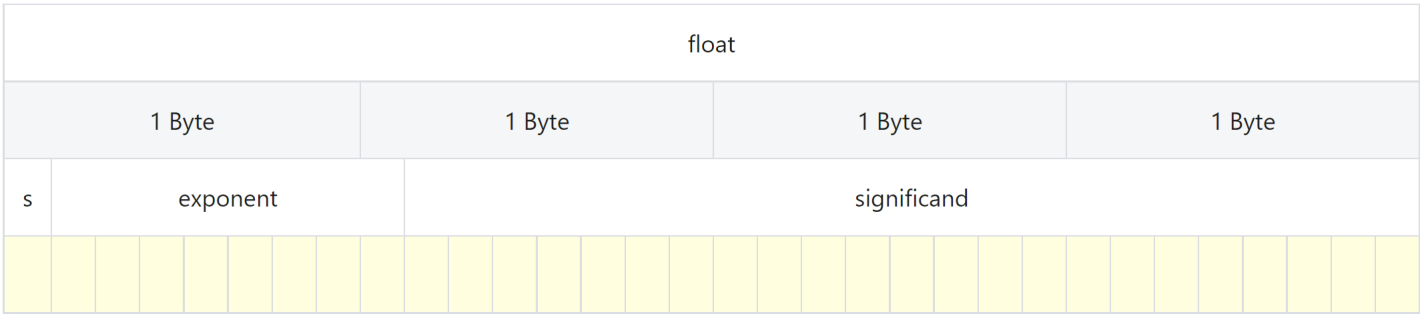
Bit #	7	6	5	4	3	2	1	0
92 =>	0	1	0	1	1	1	0	0
Flip Bits	1	0	1	0	0	0	1	1

Add 1	0	0	0	0	0	0	0	1
-92 =>	1	0	1	0	0	1	0	0

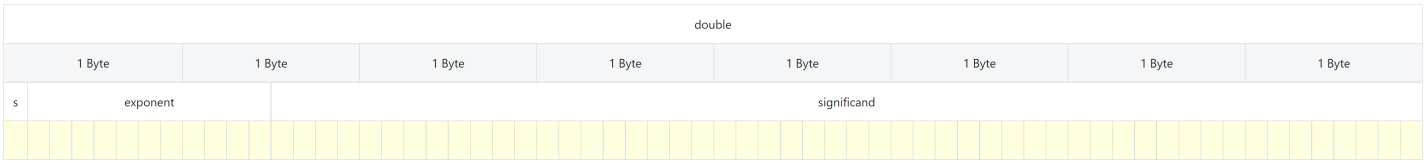
Floating-Point Data

Floating-point types store tiny as well as huge values by decomposing the values into three distinct components: a sign, an exponent and a significand. The C language leaves the implementation details to the hardware manufacturer.

The most popular model is the IEEE (I-triple-E or Institute of Electrical and Electronics Engineers) Standard 754 for Binary and Floating-Point Arithmetic. Under IEEE 754, a float has 32 bits, consisting of one sign bit, an 8-bit exponent and a 23-bit significand (or mantissa):



Under IEEE 754, a double occupies 64 bits, has one sign bit, an 11-bit exponent and a 52-bit significand:



Since the number of bits in the significand is limited, the `float` and `double` types cannot store all possible floating-point values exactly. That is, the floating-point types store values approximately.

Value Ranges

The number of bytes allocated for a type determines the range of values that that type can store.

Integral Types

The ranges of values for the integral types are shown below. Ranges for some types depend on the execution environment:

Type	Size	Min	Max
`char`	8 bits	-128	127
`char`	8 bits	0	255
`short`	>= 16 bits	-32,768	32,767
`int`	2 bytes	-32,768	32,767
`int`	4 bytes	-2,147,483,648	2,147,483,647
`long`	>= 32 bits	-2,147,483,648	2,147,483,647
`long long`	>= 64 bits	-9,233,372,036,854,775,808	9,233,372,036,854,775,807

Floating-Point Types

The limits on a float and double depend on the execution environment:

Type	Size	Significant	Min Exponent	Max Exponent
`float`	minimum	6	-37	37
`float`	typical	6	-37	37
`double`	minimum	10	-37	37
`double`	typical	15	-307	307
`long double`	typical	15	-307	307

> **NOTE**
>
> Both the number of significant digits and the range of the exponent are limited. The limits on the exponent are in base 10.

Variable Declarations

We store program data in variables A declaration associates a program variable with a type. The type identifies the properties of the variable.

In C, a declaration takes the form:

```
[const] type identifier [= initial value];
```

The brackets denote an optional part of the syntax.

We select a meaningful name for the identifier and optionally set the variable's initial value. We conclude the declaration with a semi-colon, making it a complete statement.

For example:

```
char  children;
int   nPages;
float cashFare;
const double pi = 3.14159265;
```

> **GOOD PRACTICE**
>
> It is good practice to:
>
> - declare all variables at the **beginning** of the function
> - group related variables together
>
> Following this practice will contribute towards optimal variable organization and easier to maintain code.

Multiple Declarations

We may group the identifiers of variables that share the same type within a single declaration by separating the identifiers by commas.

For example,

```
char   children, digit;
int    nPages, nBooks, nRooms;
float  cashFare, height, weight;
double loan, mortgage;
```

Naming Conventions

We may select any identifier for a variable that satisfies the following naming conventions:

- starts with a letter or an underscore (`_`)
- contains any combination of letters, digits and underscores (`_`)
- contains less than 32 characters (some compilers allow more, others do not)
- is not a C reserved word

GOOD VARIABLE NAMING TECHNIQUES

Variable names (identifiers) should...

- be self-documenting (should not require comments to describe what they are used for)
- be concise but not so short that it is cryptic
- accurately describes the data being stored
- help with the reading of the code
- use "camelNotation" (first letter of each word is capitalized with the exception of the first word)
- avoid underscore characters which are commonly used in system libraries to avoid conflicts
- **not** use underscore (`_`) characters in order to avoid conflicts with system libraries

Reserved Words

The C language reserves the following words for its own use:

```
auto        _Bool       break      case
char        _Complex    const      continue
default     restrict    do         double
else        enum        extern     float
for         goto        if         _Imaginary
inline      int         long       register
return      short       signed     sizeof
static      struct      switch     typedef
union       unsigned    void       volatile
while
```

For upward compatibility with C++, we avoid using the following C++ reserved words:

```
asm              friend           template
bool             mutable          this
catch            namespace        throw
class            new              true
const_cast       operator         try
delete           private          typeid
dynamic_cast     protected        typename
explicit         public           using
export           reinterpret_cast virtual
false            static_cast      wchar_t
```