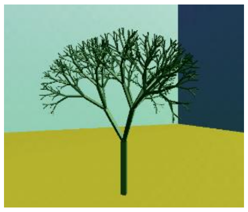

GATS Companion to C/C++ Data Types

Author: Garth Santor
Editors: Trinh Hān
Copyright Dates: 2020-01-18
Version: 0.2.0 (2020-01-31)

Overview

In binary computers, all data are a sequence of ones and zeros. Those sequences could represent letters, numbers, truth or falsehood, or anything that we choose for them to mean. The binary sequence 01000001 could be the integer 65, the character 'A', the sequence false-true-false-false-false-false-false-true, or a color in a palette. Which it is depends on the computer has been instructed to interpret those bits.

Data types emerge from mathematics, and are present in virtually ever programming environment.

In this document we will examine the concept of a data types, and how they manifest in C and C++ programming. Understanding the data type concept, and the capabilities of C and C++ data types will help you choose the right type for your solution and avoid unnecessary errors.

What is a data type?

When I ask novice programmers, "what is a data type?" I usually get the answer, "the type of a variable."

I'll usually point out that its cheating to use the word you are defining in your definition, and ask them try again without using the word 'type'. Subsequent attempt include:

- How you tell the computer what ~~type~~ kind of data you are storing.
- What the information means.
- How information is formatted.
- How information is stored.
- What kinds of things you are allowed to do with the information?

It usually concludes with, "we all know what it is – it's just hard to express!"

This always worries me, because I can't be certain that what I *know* it to be, covers all the same things that you *know* it to be.

Concepts

The concept of types can trace its philosophical origins back to Aristotle's *Categories* from the *Organon*[1]. Fortunately, we don't have to go back that far to find what we are looking for. Most of the concepts used to define computer data types come from mathematics and

VALUE

A **value** is a specific measure of quantity, or a statement of quality.

24 is value of quantity.
Yellow is a value of quality.

[1] The *Organon* is a collection of Aristotle's six works on logic.

LITERAL

A **literal** is a mathematical and programming element that expresses a single value. It cannot be changed (e.g., four is always four).

> Math: **VI** is the roman numeric literal equivalent to the Western Arabic numeric literal **6**.
> C: `1000u` is the literal representing one thousand as 32-bit unsigned integer.
> C++: `"hello"s` is the `std::string` literal for the word 'hello'

DATA

A **datum** (plural: **data**) is an individual fact, or item of information. A datum is a value in a context.

> The width of a soccer goal is **24** feet.
> The colour of sulfur is **yellow**.

VARIABLE

A **variable** is a mathematical element which functions as a placeholder for a value. In programming, it is an element that represents the storage location of a value.

> Math: $Let\ x \in \mathbb{Z}$ is a statement declaring that x is an integer.
> C: `int x = 42;` is a statement declaring x to be a signed integer initially storing the value 42.

REFERENCE

A reference is a value that enables a program to indirectly access a particular datum in a computer's memory[2]. Reference can be called *references*, *pointers*, or *handles* in programming languages. The differences between a variable and reference can be subtle.

```
C/C++ variable:   int n = 42;  //n is an integer variable containing the value 42.
C/C++ pointer:    int* p = &n; //p is a pointer containing the address of variable 'n'.
C++ reference:    int& r = n;  //r is a reference to the same location as variable 'n'.
```

The variable 'n' holds the integer value.
The pointer 'p' holds the address-of the of variable 'n'. Accessing the value of 'n' through 'p' requires an extra step.
The reference 'r' refers to the same location as variable 'n'. If the declaration of 'r' and 'n' are in the same scope – they are effectively 'both' the variable. If 'r' is assigned a variable as function parameter, it is effectively a pointer and has the extra step to access what it points to.

Variables' locations are typically determined at compile time at either an absolute location in memory (global or static allocation), or as a fixed offset from the top of the stack (local/automatic allocation).

Reference variables declared in the same scope as what they reference, are determined at compile time, whereas reference parameters (C++ only) are assigned at runtime.

Pointer variables are almost always assigned at runtime, as the location a pointer references can change (unlike a C++ reference).

TYPE

A **type** is the name given to a number of things sharing one or more characteristics that cause those things to be regarded as a group, class, or category.

> *Cheese* and *ice cream* are both **types** of *dairy products* as they share the characteristic of being made from the milk of mammals, whereas *soy milk* it not a dairy product as it is made from a plant.

[2] Reference (computer science) - Wikipedia

SET

A **set** is a collection of objects (things, people, ideas, etc.) For mathematical, and primitive computing types, the objects are usually numbers.

> In C, a **short int** may hold any value from the set of integers from $-32{,}768$ to $+32{,}767$.
> In C++, a **bool** may hold either value of **false** and **true**.

To say, "**42 belongs to the set *S***", we write "$\mathbf{42 \in S}$".

To say, "***D* is a set composed of the values 42, 3, and 5**", we write "$\boldsymbol{D = \{3, 5, 42\}}$". Note that the order of elements in a set does not matter.

We often use the word "**is**" to express set membership. For example, the statement "**7 is a prime number**" is to say "**7 belongs to the set of prime numbers**".

FUNCTIONS

Functions take a value of one type, and turn it into a different value of the same or another type. Functions can be identified grammatically as a noun phrase.

> *The answer is the square of four.* "**the square of four**" is a noun phrase that could be replaced by the value **sixteen** without changing the meaning of the sentence. The equivalent statement is, "*The answer is sixteen.*" The phrase, "the square of" plays the role of a function transforming the number 'four' into the number 'sixteen'.

RELATIONS

Relations describe how thing associate with or compare to each other. Common relations include:

- Equivalence relations (are two things the same). E.g., equality, inequality.
- Ordering relations (which thing comes first). E.g., less than, greater than.
- Containment relations (which thing can be found within the other). E.g., contains.

BINARY OPERATIONS

Binary operations describe how values of a data type can be combined to form a value of same type or a value of a different type.

> The addition of two natural numbers produces a natural number $(40 + 2 = 42)$.
> The division of two natural numbers may produce a natural number or a quotient ($42 \div 7 = 6$ and $42 \div 5 = \frac{42}{5}$).

Mathematical view

Mathematicians describe data types by examining how they related to each of four different concepts.

- Sets: what values are allowed;
- Functions: what transformations are allowed;
- Relations: what comparisons are allowed;
- Binary operations: how can values be combined to make new values.

Example: Boolean

> Set: $B = \{\, false, true \,\}$
> Functions: not
> Relations: equality, inequality
> Binary operations: and, or, xor, nor, nand, xnor

Computational view

Computer science adds to the mathematical description of data type by including its limitations. In computing data types are typically finite (they have minimums and maximums), and consume resources (time and space). A description of the type's representations become important in understanding those limitations.

- How memory efficient is the data type?
- How computationally efficient is the data type?
- Can it be implemented efficiently on the necessary platform, or language?

Example: a 4-byte integer can be represented using:

- A character sequence of digits (4 digits without sign)
- Binary coded decimal (8 digits without sign)
- Sign + magnitude (1 sign bit, 2^{31} magnitude bits, adding positives and adding negatives use different hardware, two representations for zero)
- One's-complement (negatives are the bit-inverse of positives, single hardware for adding positive and negative numbers, two representations for zero)
- **Two's-complement** (like one's-complement, but only a single representation for zero). *This is the typical hardware implemented representation*.

My Definition

A data type is the name given to a programming element that:

- Defines a set of values;
- Defines a set of binary operations and their properties that work with those values;
- Defines a set of relations for those values;
- May define a set of functions of those values;
- May define a representation of those values.

Categorizing data types

Primitive vs. composite

Primitive data types are typically the data types 'built-in' to the language. Characteristics common to primitive types:

- There are no language elements below them. They are not made up of other things that the language allows you to manipulate. Operations apply to there whole, not to a part (i.e., you can't add just part of a **double**).
- Their values can be represented by literals.
- They are the smallest allocatable unit in a programming language (i.e., you can't allocate half of a character).

Composite data types are derived from more than one primitive type. Languages provide mechanism to create compound data types including:

- Arrays
- Records / Structures
- Unions
- Sets
- Objects

Language	**Primitives (selected)**
Java	`byte, short, int, long, float, double, boolean, char`
C	`void, char, short, int, long, long long, _Bool, unsigned short, float _Complex, double _Imaginary,` *pointer*
C++	`void, char, short, int, long, long long, bool, unsigned short, char32_t,` *pointer*, *reference*
Language	**Compounds (selected)**
Java	*Array*, `String, Set, List, Map, …`
C	*Array*, `struct, union`
C++	*Array*, `struct, union, class, pair<>, tuple<>, vector<>, string, …`

Machine type vs. software data types

Machine data types are understood directly by the hardware. The CPU's instruction set contains operations to manipulate data of that type.

"IMUL – Signed Multiply" is an assembly (machine) instruction for multiplying 2's-complement integers on x86 CPUs. Parameters to the instruction indicate whether the operation applies to 16, 32, or 64-bit values.

"**`std::string`**" is a software data type in C++ for character sequences. In practice there have been several different implementations that all manifest the same behaviours.

The primitives of C and C++ are generally machine types (`int`, `double`, `char`, *pointers*) or directly mappable to a single machine type (e.g., the C99 type `_Bool` maps to the machine type `int8_t`).

Abstract vs. concrete data types

A data type is abstract if it does not specify the representation, concrete if it does.

The integer data type in Python is effectively abstract as its doesn't expose its representation. Python 3's integer type actually switches between two representations: the machine type when the value is within the machine type's range, and a different dynamic type, when beyond the machine type's range.

The **short int** data type in C is concrete. It must have a 16-bit, 2's-complement representation, which is the standard two-byte machine type of the CPU.

Mathematical Data Types

Why bother with mathematical data types? They are the basis of the numerical data types in programming. The most important data types in mathematics are:

Natural Numbers

Natural numbers are the set of numbers starting at either zero or one and increasing to infinity. They represent the *natural counting numbers*. There is still some debate as to whether zero should be included in the set of natural numbers.

The set of natural numbers is denoted by the double-stroke capital letter ℕ (U+2115)[3]. Unfortunately, this is ambiguous as to whether zero is included in the set. Where the inclusion or exclusion of zero is critical we employ subscripts 0 or superscript *.

$\mathbb{N}_0 = \mathbb{N}^0 = \{0,1,2,3,\cdots,\infty\}$

$\mathbb{N}^* = \mathbb{N}^+ = \mathbb{N}_1 = \mathbb{N}_{>0} = \{\,1,2,3,\cdots,\infty\}$

[3] In the absence of the font *blackboard bold*, Helvetica bold is typically used

Integers

Integers are the set of numbers including all the *natural numbers*, *zero*, and the *negatives* of all the *natural numbers*. The addition of negatives allows integers to represent differences that are both positive and negative.

The set of integers is denoted by the letter ℤ (U+2124). It may seem odd to a speaker of English that **I**ntegers are denoted by a ℤ. The ℤ, stands for *Zahlen*, a German word for "numbers". Subsets of integers can be denoted with superscripts:

$\mathbb{Z} = \{-\infty, \cdots, -3, -2, -1, 0, 1, 2, 3 \cdots, \infty\}$

$\mathbb{Z}^{+} = \mathbb{N}$

$\mathbb{Z}^{\geq} = \mathbb{N}^{0}$

$\mathbb{Z}^{\neq}$ for non-zero integers (i.e., $\{-\infty, \cdots, -3, -2, -1, 1, 2, 3 \cdots, \infty\}$)

Quotients/rational numbers

Commonly called fractions, the quotients or rational numbers are a composite number composed of two integers. One number indicates the *number of parts* (numerator), and the other indicates how many parts make a whole (denominator). We write the numerator above the denominator separated by a horizontal line.

$$\left.\begin{matrix} 1 & \leftarrow \text{dividend or numerator} \\ \overline{2} & \leftarrow \text{divisor or denominator} \end{matrix}\right\} \leftarrow \text{quotient}$$

The set of quotients is denoted by the letter ℚ (U+211A).

Any integer is possible for either numerator or denominator with the exception of zero as the denominator.

$$\mathbb{Q} = \left\{\frac{n}{d}\right\}, where\ n \in \mathbb{Z}\ and\ d \in \mathbb{Z}^{\neq}$$

Irrational numbers

Irrational numbers are all the real numbers that cannot be expressed as a ratio of two integers. π, the ratio of a circle's circumference to its diameter is an irrational number. Irrational numbers can be written in positional notation (a number with digits following the decimal place), but the decimal expansion does not terminate, nor end with a repeating sequence.

Irrational:	$Pi = \pi = 3.14159\ldots$	this goes on forever, without repetition
Rational:	$\frac{4}{3} = 1.333\dot{3}$	this goes on forever, but repeats

Important irrational numbers:

$Pi = \pi = 3.14159\ldots$	ratio of a circle's circumference to its diameter
$Euler's\ number = e = 2.71828\ldots$	base of the natural logarithm
$Golden\ ratio = \varphi = \frac{1+\sqrt{5}}{2} = 1.61803\ldots$	common ratio in natural, finance, etc.
$Root\ of\ 2 = \sqrt{2} = 1.4142135\ldots$	architecture and geometry

Real numbers

Real numbers are values that can represent a distance along a line (a real-world measurement). Real numbers include all the rational numbers or quotients (and therefore all the integers and natural numbers), and all the irrational numbers (and therefore transcendental numbers).

Real numbers are commonly represented using decimal notation where the digits following the decimal separator represent the numerator of a fraction whose denominator is $10^{\#digits\ in\ numerator}$.

$$12.345 = 12\frac{345}{1000} = 12\frac{345}{10^{3}}$$

The decimal separator is typically a period (.) in English-language countries such as UK, USA, and Canada, but a (,) in continental European countries such as Germany and France.

$$12.34 \ in\ Canada = 12{,}34 \ in\ Germany$$

The symbol ℝ (U+211D) is used to denote the set of real numbers.

Complex numbers

Complex numbers allow us to solve an immense collection of problems in geometry, signal analysis, physics, and quantum mechanics (too name just a few). They are composed a what we call a *real* part and an *imaginary* part – the imaginary part indicated by the constant i, where $i^2 = -1$ or $i = \sqrt{-1}$.

The symbol ℂ (U+2102) is used to denote the set of complex numbers.

$$\mathbb{C} = \{a + bi\}, where\ a, b \in \mathbb{R}, and\ i = \sqrt{-1}$$

How the number systems relate to each other

Each number system (type) is contained within another.

$$\mathbb{N}_1 \subset \mathbb{N}_0 \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$$

C/C++ Primitive Data Types

Arithmetic types – floating point

Floating-point types are used to represent **real numbers**.

Programmers must be aware that they cannot represent every number precisely (in-fact, most are approximations).

Mantissa (or significand) is the part of a number in scientific notation consisting of the significant digits. The exponent indicates how large or small the number is.

In the number $234{,}000 = 2.34 \times 10^5$, 2.34 is the mantissa, and 5 is the exponent.
In the number $0.0125 = 1.25 \times 10^{-2}$, 1.25 is the mantissa, and -2 is the exponent.

C and C++ share the following floating-point types:

Name	Size	Value Range	Precision	Exponent	Mantissa
• `float`	4 bytes	Smallest: $\pm 1.175494351 \times 10^{-38}$ Largest: $\pm 3.402823466 \times 10^{+38}$	6-7 decimal digits	8 bits	23 bits
• `double`	8 bytes	Smallest: $\pm 2.2250738585072014 \times 10^{-308}$ Largest: $\pm 1.7976931348623158 \times 10^{+308}$	15-16 decimal digits	11 bits	52 bits
• `long double`	≥8 bytes	≥ double			

The long double type is only required to be at least a double. Compilers are free to implement the type differently.

Compiler	Precision
Microsoft C++ on x86	Same as double (64-bit)
Intel C++ compiler on Windows GNU C compiler or x86	80-bit
HP-UX Solaris/SPARC	128-bit

DECLARATIONS

-
  ```
  float simplePi = 3.14159F;
  double standardPi = 3.141592653589793;
  long double precisePi = 3.14159265358979328846264433L;
  ```
-

SPECIAL VALUES

Floating-point types can represent INFINITY which is the result of division by zero, and NaN (Not-a-number) which is the result of computing the square root of a negative number.

Arithmetic types – integer types

Integer types are declared with the

‘`int`’ types are used to represent **integer numbers** and are stored as 2’s-complement binary numbers.

‘`unsigned int`’ types are used to represent **natural numbers** and are stored as straight binary numbers.

Unfortunately, C/C++ integer types don’t behave the same way on all platforms (unlike Java[4]) as they try to efficiently map C/C++ types to machine data types, but not all CPUs support the same machine types. This leads to a lot of confusion and difficulty in portability. C/C++ also complicates things by assigning multiple names to the same type.

What they do specify is the names and aliases and ranking of the types and the minimum bits for each type.

Ranking: the order of the types from smallest to largest. In C/C++ the ranks can’t overlap.

Let’s start with the ISO rules on the integer types…

Equivalent Type (the prototypical or easiest name)	**Width in bits**	**Type specifier/aliases** (what you call it in the code)

`char`	`≥ 8 or smallest type to store a member of the basic execution set.`[5]	`char`
		`signed char`

`unsigned char`	`≥ 8 or smallest type to store a member of the basic execution set.`	`unsigned char`

`short int`	`≥ 16 and ≥ char-bits`	`short int`
		`signed short int`

`unsigned short int`	`≥ 16 and ≥ char-bits`	`unsigned short`
		`unsigned short int`

`int`	`≥ 16 and ≥ short int`	`int`
		`signed`
		`signed int`

`unsigned int`	`≥ 16 and ≥ unsigned short int`	`unsigned`
		`unsigned int`

`long int`	`≥ 32 and ≥ int`	`long`
		`long int`
		`signed long`
		`signed long int`

`unsigned long int`	`≥ 32 and ≥ unsigned int`	`unsigned long`
		`unsigned long int`

`long long int`	`≥ 64 and ≥ long int`	`long long`
		`long long int`

[4] Java keeps it simple and dictates that all JVMs must implement all types in exactly the same way.
[5] ‘`char`’ is the minimal character set that language is expressed in (usually ASCII).

		`signed long long` `signed long long int`
`unsigned long long int`	`≥ 64 and ≥ unsigned long int`	`unsigned long long` `unsigned long long int`

Implementation

The actual implementation of the integer types can vary by compiler and by platform, so long as the preceding rules are followed. The choices about implementation sizes are called the *data model* and can be described by identifying the size choices for int, long, and pointers. Why pointers? Because they are part of the integer types. Pointers are integers that identify a specific memory location (the address) by its offset from the beginning of RAM. Pointers are implemented as the unsigned integer type that can index every location in memory, but is no larger than is necessary.

Data Models

The four most common data models for C/C++ are LP32, ILP32, LLP64, and LP64.

Model	Platforms	int	long	pointer
LP32	Win16 API	16	32	32
ILP32	Win32 API 32-bit Unix, Linux, macOS	32	32	32
LLP64	Win64 API	32	32	64
LP64	64-bit Unix, Linux, macOS	32	64	64

Common Implementations

Type	MSVC (x86)	MSVC (x64)	GNU C (Intel 64 bit)
• `char`	8 bits	8 bits	8 bits
• `short`	16 bits	16 bits	16 bits
• `int`	32 bits	32 bits	32 bits
• `long`	32 bits	32 bits	64 bits
• `long long`	64 bits	64 bits	64 bits
• `intptr_t/uintptr_t`	32 bits	64 bits	64 bits
• `size_t`	32 bits	64 bits	64 bits
• `intmax_t / uintmax_t`	64 bits	64 bits	64 bits

Data Ranges

Bits	Bytes	Format	Minimum	Maximum
• 8	1	magnitude 2's complement	0 –128	• `+255` • `+127`
• 16	2	magnitude 2's complement	0 –32,768	• `+65,535` • `+32,767`
• 32	4	magnitude 2's complement	0 –2,147,483,648	• `+4,294,967,295` • `+2,147,483,648`
• 64	8	magnitude 2's complement	0 – 9,223,372,036,854,775,808	• `+18,446,744,073,709,551,615` • `+9,223,372,036,854,775,807`

Declarations

-
  ```
  short smallNumber = 3;          // There is no literal specifier for short
  int n = 42;
  long bigger = 1000000000L;      // billion
  ```

-
  ```
  long long huge = 1000000000000LL;   // trillion
  ```

- `uint64_t giantSize = 42LLU;         // literal is unsigned long long`

Special types

size_t

When computers moved from 16-bit to 32-bit, ‘`int`’ and ‘`unsigned`’ moved with them. But when computers moved from 32-bit to 64-bit, making ‘`int`’ 64-bit would break the rule that it never exceeds the size of a ‘`long`’. The solution was to introduce a new data type specifically to handle the size values of arrays. Instead of adding a new primitive type, they used `typedef`s to create a switching type.

`size_t` is defined in the header file `<stddef.h>` in C, or `<cstddef>` in C++. On 32-bit and smaller systems it compiles to an `unsigned long integer`. On 64-bit and larger systems it compiles to an `unsigned long long integer`. This ensures that size_t will always be large enough to handle any memory address or offset for that system, but not be so large as to be wasteful.

intmax_t, uintmax_t

`intmax_t` and `uintmax_t` are defined in the header files `<stdint.h>` in C, or `<cstdint>` in C++. They map to the integer types best suited to hold a pointer on that platform. On some systems, this can mean that `intmax_t` is actually 128-bits!

intptr_t, uintptr_t

`intptr_t` and `uintptr_t` are defined in the header files `<stdint.h>` in C, or `<cstdint>` in C++. They map to the largest integer types supported on that platform.

Boolean Types

The original C did not implement a Boolean type as there is no Boolean type implemented in hardware. CPUs handle Booleans via bit flags in a register. To store an individual Boolean ‘bit’, a programmer would have to store the value in a byte or a bit field, neither of which are understood by the CPU.

What then, do we do? The convention is to use an int as Boolean type as the compiler generates code that will interpret the integer value of zero as a Boolean false, and non-zero as Boolean true. The following code:

-
  ```
  double x = 3.1, y = 4.2;
  int b = x < y;
  if (b) …
  ```

will store the value 1 in variable ‘b’. The ‘if’ statement will then execute the body of the conditional statement since ‘b’ is non-zero.

The merits of the approach are that Boolean tests can be performed without first having to convert to a Boolean representation. This results in the code:

-
  ```
  int* p = malloc(n);
  if (p != NULL) {
      // use the pointer
  }
  ```

can be written idiomatically as:

-
  ```
  if (int* p = malloc(n)) {
      // use the pointer
  }
  ```

Here, the pointer is declared within the scope of the if-statement, and assigned address produced by the function ‘malloc’. If the pointer receives a non-NULL value, the conditional is executed with the pointer, if malloc returns a NULL pointer, the condition is exited and the pointer goes out of scope.

_Bool

C99 introduced the new primitive data type `_Bool`. The odd name was given to ensure that the name doesn't interfere with existing code. The `_Bool` type behaves like the earlier int – zeros are false, non-zeros are true. The difference is that `_Bool` generates code to ensure that 'true' is only stored as `(int)1`. The storage requirement for `_Bool` is only required to be large enough to hold the values 0 or 1, so most implementations use a byte.

bool

C++ implemented a Boolean data type primitive early on with the keyword `bool`. It too produces machine code to restrict 'true' to being stored only as `(int)1`. C++ also defines keywords for the literals `true` and `false`.

Since lower-level C++ code is often shared with C, having two different Boolean implementations is an irritant!

C99 supplies `<stdbool.h>` which defines `bool`, `true`, and `false` as macros. C++ provided `<cstdbool>` which is the C++ equivalent to `<stdbool.h>` but it was deprecated in C++17 and removed in C++20. Use `<stdbool.h>` when sharing code between C and C++.

Best practices…

< C99

- Use int as the Boolean data type as the default choice. It is the idiomatic choice, and the easiest for the compiler to optimize.
- Use a char as a Boolean data type when you want to conserve space.
- Don't write your own macros for bool, true, and false. You will not be able to simulate the all trues are stored-as-one behaviour so it is better to keep the Booleans distinct. Try a distinct identifier like:

```
typedef int BOOLEAN;
#define BTRUE 1
#define BFALSE 0
```

  This will also make it much easier to convert the code to the standard bool in the future.

≥ C99 Use `<stdbool.h>` for all new code. Upgrade as opportunities arise. Compilers and linters have learned about `<stdbool.h>` and can perform additional optimizations and safety checks.

C++ Use the primitive bool data type – the standard does!

In the next update…

You'll note that this is version 0.2.0 – it's still under development. But check back, I intend to add information on:

- Characters
- Strings
- Programming pitfalls
- Best practices

Document History

Creation	0.10: 2021-01-18 {concepts + numeric types} 0.20: 2021-01-31 {added memory models to numeric types}