

Types (I)

Overview

- Types determine **how a sequence of bits** is to be **interpreted by the compiler**. The format of some types are common across all platforms, while others depends on the platforms.
 - In some cases, such as **floating point** representations, the format and operations are defined as a standard (IEEE standard) so that the data type is common **across all platforms**.
 - In other cases, such as the **int or long** representations, the data type is **platform and compiler dependent**.
 - **Integer** data types are a **natural internal representation** because they have a **trivial mapping** into a binary representations, and **manipulation** on binary numbers is **well defined**. (2's complement)
 - The assumption that **integer representations and manipulations are natural** has **led to** the odd fact that **integer formats and manipulations are not common across computer across all languages**.
 - **C** lets the **compiler define the format for the int and long** data types **based on** the machine **hardware**. While this has the **benefit** that it **optimizes memory usage and integer arithmetic** for the hardware, it also means the code that works on one machine may not work on another.
 - Many programming languages **eliminate this problem** by **defining a specific integer format** as part of the language. **Java**, for example, are **portable** because their type formats are defined in the language specification.
- A type also defines the **set of operations** applicable to the data.
 - E.g., Python allows + and * apply on basic types and also string. When applying * on string, the right-hand-side of the operator must be an integer. But - and / can apply on basic types only.
- Statically typed vs dynamically typed
 - Variable **types** do **not** have to be **assigned at compile time** or assigned **explicitly**.
 - **Statically typed** if the variable types are determined or specified at **compile time**. (C, Java)
 - **Dynamically typed** if the variable types are determined or specified at **run time**. (Python)
- A **type error** is when a program executes **an operation on a data type** for which the operation **is undefined**. (e.g., string1 + string2 in C)

- Type errors can have significant consequences, because they affect very low level operations that algorithms depend on being correct.
- Strongly typed vs weakly typed
 - **Strongly typed**: stricter typing rules at compile time and runtime
 - **Weakly typed**: looser typing rules, may produce unpredictable results or may perform implicit type conversion at runtime

Formats

- Programming languages do not always fully define the semantics of data types.
 - Java defines data type sizes explicitly, because it has control over the architecture.
 - C and Python do not.
 - Python is implementation dependent because it is built on C. In Python, integers have the same size as floating-point values: 8 bytes. Although Python can now gradually increase the size for large int values, compared with C, it's less memory-optimized.
- The main reason for defining a data type as part of a language specification is portability of code.
- The main reason for not defining it is optimization to the particular architecture.
- A data type that has no hardware support may be convenient, but it will also be slower.
- The data type size represent the range a data type can represent.

Data Type	Size (Byte)	Range	2' complement	Printf Format
char	1	-128 to 127	-2^7 to 2^7-1	%c
unsigned char	1	0 to 255	0 to 2^8-1	%c
short	2	-32,768 to 32,767	-2^{15} to $2^{15}-1$	%hd
unsigned short	2	0 to 65,535	0 to $2^{16}-1$	%hu
int	4	-2,147,483,648 to 2,147,483,647	-2^{31} to $2^{31}-1$	%d
unsigned int	4	0 to 4,294,967,295	0 to $2^{32}-1$	%u

Implicit conversions

- Implicit conversions between types are a common occurrence in most programs and programming languages.
- Most languages have a set of rules for implicit conversions that may occur without producing a warning, even strongly typed languages.
 - [show conversions.c] The ANSI standard for C defines the set of implicit conversions for numeric types in expressions with binary operators. The idea is to [keep the precision after conversion](#). (float → double, small size → large size, widening conversion)

```
/**
 * implicit conversion (overflow)
 * conversions.c
 * Ying Li
 */

#include <stdio.h>

int main () {
    short i = 3;
    int j = 2147483647;
    short s = i + j;
    printf("short i %d, int j %d, short s %hd \n", i, j, s);
    int n = i + j;
    printf("short i %d, int j %d, int n %d \n", i, j, n);
    unsigned int m = i + j;
    printf("short i %d, int j %d, unsigned m %u \n", i, j, m);

    long x = 2147483647;
    long l = i + x;
    printf("short i %d, long x %ld, long l %ld \n", i, x, l);

    return 0;
}
```