CS 315 – Programming Languages
Types

<u>Uses of types</u>

- They provide implicit context for operations
  E.g.: x / y; // integer or floating point division depending on the types of x and y
- They limit the set of operations that can be performed on variables
  E.g.: x + 3; // not valid if x is a string, valid if x is an integer
- They limit the set of values that can be stored in variables
  E.g.: x = "ab"; // valid if x is a string, not valid if x is an integer

<u>Type checking</u>

- *strongly typed languages*: prohibits application of any operation to an object that is not intended to support that operations
- *unsafe languages*: otherwise

We can also classify type checking based on when the check is performed:

- *statically typed*: mostly checked at compile-time (which means you have to specify the type of the variable while declaring it)
- *dynamically typed*: mostly checked at run-time

<u>Type equivalence</u>

- name-based equivalence: if two types have the same name, then they are equivalent
- structural equivalence: if two types have the same structure, then they are equivalent

E.g.: C uses name-based equivalance:

```
struct X1 { int a; int b; };
struct X2 { int a; int b; };
```

These are not equivalent in C, even though they have the same structure.

→ a and b has to have the same decleration order

<u>Type conversions</u>

```
x = y * z;
x + y;
foo(x, y, z);
void foo(T x, U y, V z) {}
```

If types are same, then there are no issues. But if the types are different, then *conversions* happen.

- *Cast*: an explicit conversions
  E.g. in C: int x = 2; double y = 3.5; x = x * ((int) y); // y is cast into an integer
- *Coercion*: implicit conversion of one type to another
  E.g. in C: int x = 2; double y = 3.5; y = x * y; // in x * y, x is coerced into a double

<u>Conversion scenarios</u>

- Types are structurally equivalent but language uses name-based equivalence X2 from the previous example would have to be converted into X1
- Types have intersecting values
  E.g. in Java (casting): float x = 8; int y = (int) x;
  E.g. in Java (coercion): short x = 3; long y = x;

Coercion
Can happen in assignments, arithmetic operations, function calls

int a = 5;
double x = a;
// a is an int, but it will be coerced into a double
	0.1 / 5
	// 0.1 is a double
	// 5 is an int, but it will be coerced into a double
		void foo (float x) { ... }
		int y; foo(y);
		// y is an int, but it will be coerced into a float

Coercion in C via examples:

```
int16_t s;
uint32_t l;
float f;
double d;

s = l; // least significant bits of l are taken
l = s; // s is sign extended and the interpreted as unsigned
f = l; // l is converted to floating point, some precision might be lost
d = f; // f is converted to double precision, no precision is lost
f = d; // d is converted to single precision, precision can be lost
```

Universal reference types

C → void * – raw memory
Java → Object – has common properties
ElllieLong → Ellie – has common shape properties.

Types can be divided into broad categories:

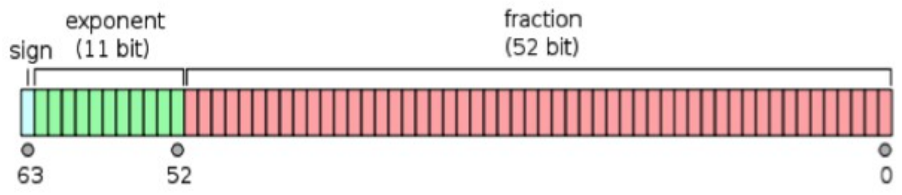
- *primitive*: they are not defined in terms of others
- *composite*: defined in terms of other types

<u>Common primitive types</u>

- Fixed-size integer types:
  - Java: byte (1), short (2), int (4), long (8)
  - C++: short, int, long may have platform specific size, int8_t, int16_t, int32_t, int64_t have well defined sizes
  - Java: Unsigned types are not supported, as they are a common source of mistakes:
    - In C++: for (size_t i=k; u>=0; --i) {…} // size_t is unsigned, results in infinite loop

guaranteed to be big enough for any array index.

  - C++: unsigned types are supported, as they give a larger range, and are used in low level programming, such as network protocol code. E.g.: unsigned int, uint32_t
  - These are all fixed-size integers, that map closely to the hardware. Operations on them are supported by the hardware and generally fast.
- Arbitrary sized integer types:
  - Most languages support these as well, such as BigInteger in Java. These are implemented in software and could be slow. Here, slow also has to do with asymptotic complexity. A fixed-size integer operation always takes the same time irrespective of the contents of the values involved. However, for arbitrary sized integer types, such as BigInteger, the larger the values, the slower the operations are. Larger values are represented with more bytes in memory and operations on them has to process more data, resulting in more work.
- Floating point types
  - These can be classified as:
    - Binary floating-point
    - Decimal floating-point
    - Fixed-point decimal
    - Arbitrary-precision decimal
  - Binary floating-point:
    - Defined by IEEE floating-point standard 754
    - Two kinds: single-precision and double precision



The real value assumed by a given 64-bit double-precision data with a given biased exponent e and a 52-bit fraction is

$$= (-1)^{\text{sign}}(1.b_{51}b_{50}...b_0)_2 \times 2^{e-1023} \text{ or more precisely: value} = (-1)^{\text{sign}}\left(1+\sum_{i=1}^{52} b_{52-i}2^{-i}\right) \times 2^{e-1023}$$

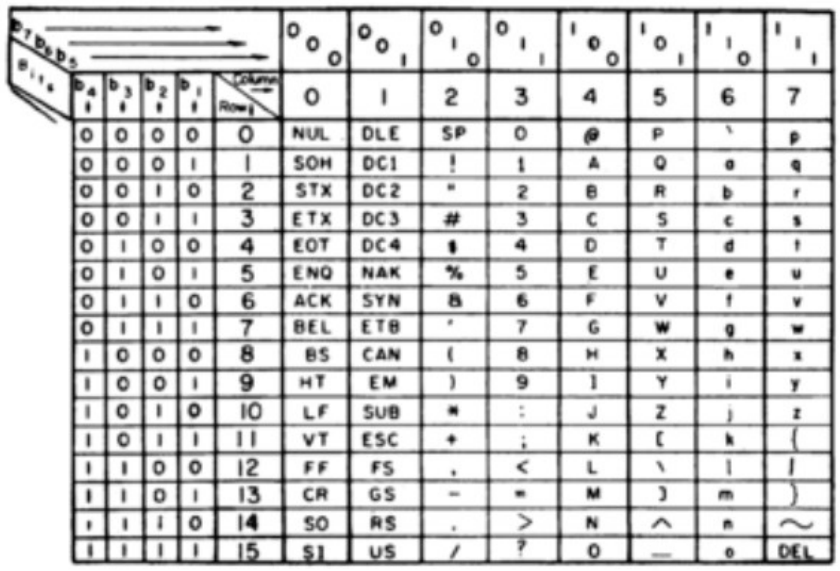
    - Cannot represent some numbers exactly, such as 0.1, as it uses binary rather than decimal representation. Binary representation is easier for the hardware to implement and most processors have a floating-point unit that perform arithmetic operations on binary floating-point variables in hardware.

        - Numerical stability: Consider summing up values. If you add a very big value to a very small one, the fractional part of the representation won't have enough bits to store the small value, and thus precision would be lost. If you are summing several values, it is best to go from small values to large ones.
    - Decimal floating-point
        - Similar to binary floating-point, but uses decimal rather than binary base
        - Typically implemented in software, but some processors implement it in hardware as well (such as IBM Power6 processor)
    - Fixed-point decimal
        - Fixed number of digits after the dot
        - Typically implemented in software
    - Arbitrary-precision decimal
        - Such as BigDecimal in Java. These are implemented in software and could be slow.
- Boolean
    - Possible values: true or false
    - Often implemented as a byte (as bits are not individually addressable)
- Enumerations
    - All possible values, which are named constans, are specified in the definition
      E.g.: C++11
      ```
      enum class Day { mon, tue, wed, thu, fri, sat, sun };
      Day d = Day::mon;
      ```

<u>Composite types</u>

- Strings & characters
    - A string is a sequence of characters
    - How do you store the size of the string?
        - As a separate variable of type integer
            - Adv: You can implement the length() operation very fast
            - Dis: It takes too much space for short strings
        - You do not store it, but end your string with the null character ('\0')
            - Adv: Very little space overhead (just one extra byte)
            - Dis: Takes O(n) time to implement the length() operation
    - How do you represent the characters
        - Depends on the encoding: how values map to the logical characters
        - ASCII is a 7-bit character-encoding scheme

USASCII code chart



b7 b6 b5 →					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b4	b3	b2	b1	Row ↓ Column →	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	−	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	—	o	DEL

        - ASCII is designed for the English language, cannot represent characters from other languages
        - Unicode is a standard for the consistent encoding, representation and handling of text expressed in most of the world's writing systems.
        - Unicode encodings
            - UTF-32: Each character is 32-bits
                - Dis: Takes up too much space, 4 times that of ASCII for English text
                - Adv: All characters are the same size, which makes indexing/slicing easy
            - UTF-16: Each character is one or two code units, where each code unit is 16-bits
                - Characters in the Basic Multilingual Plane (BMP) cover most languages and take a single code unit
                    - E.g., ğ takes 1 code unit, but 𝄞 takes 2 code units
                - Adv: In terms of space, it is better than UTF-32
                - Dis: Indexing and slicing is trouble for strings that have characters taking 2 code units
            - UTF-8: Each characters can be 1 to 6 code units, where each code unit is 1 byte
                - It is fully ASCII compatible, in the sense that an ASCII string is also a valid UTF-8 string
                    - E.g., g takes 1 code unit, ğ takes 2 code units, but 𝄞 takes 4 code units
                - When the 8th bit that is not used by ASCII is 0, then the character is 1 byte, otherwise, it could be longer
                - Adv: ASCII compatibility, small space overhead for common text
                - Dis: Indexing and slicing is difficult due to variable-sized encoding
            - Unicode has more than just encodings: collation (ordering), capitalization, normalization/decomposition, etc.
- Arrays
    - Sequence of homogeneous data elements
    - Indexing: a mapping from indices to elements
    - Range checking for indexing

ArrayOutOfBoundsException

        - Brings additional runtime cost, but is safer (Java has them, C does not)
    - Slicing: Taking a subset of the array
        - Could be copy-based or alias based (see Homework 1)
    - Multi-dimensional arrays: matrices, tensors, n-dimensional arrays
    - Heterogeneous arrays: each element could be of different type
        - E.g.: Python

```
x = [“abc”, 4, 4.5]
```

- Associated arrays – HashMaps
    - Similar to arrays, but enable indexing via keys
    - These are often implemented as hash tables
    - E.g.: Dictionaries in Python

```
x = { 'Hasan' : 14, 'Ali' : 40, 'Aslı' : 51 }
print x['Hasan']
```

- Unions
    - A type whose variables are allowed to store different type values at different times during execution
    - Saves space. The size of a variable of union type is equivalent to the size of the largest type the union stores.
    - Free vs discriminated unions
        - Free unions in C/C++

```
union Token {
    int intValue;
   char * stringValue;
   double floatValue;
};
Token token;
token.intValue = 5;
token.stringValue = “abc”;
token.floatValue = 3.5;
```

        - With free unions, there is no way to determine which type of value is currently the active one.
        - With discriminated unions, there is an attribute that specifies the currently active value
        - Discriminated unions in Ada

```
type Shape is (Circle, Triangle, Rectangle);
type Colors is (Red, Green, Blue);
type Figure (Form: Shape) is record
   Filled: Boolean;
   Color: Colors;
   case Form is
      when Circle =>
         Diameter: Float;
      when Triangle =>
         Leftside, Rightside: Integer;
         Angle: Float;
      when Rectangle => Side1, Side2: Integer;
   end case;
end record;
```

a struct with an int and char inside.

- Records
  - Possibly heterogeneous aggregate with named fields
    - e.g., structs in C, classes and structs in C++, classes in Java, etc.
- Pointers
  - Hold addresses of other variables
  - Dangling pointers: A pointer that points to (invalid) non-allocated memory
    E.g.: C++
    ```
    int* x = new int;
    delete x;
    *x = 5; // x is dangling
    ```
    this will still work but memory space used by x has been given back. When another program allocates to the space which x writes to; x will change.
  - Memory leaks: Allocated memory whose address is lost
    E.g.: C++
    ```
    void foo() {
        int* x = new int;
    }
    foo();
    // no way to reach the allocated integer anymore
    ```
    E.g.: C++
    ```
    int * x = new int;
    x = new int;
    // no way to reach the first allocated integer anymore
    ```
  - Garbage collection: automated management of deallocation of heap allocated variables
    - Reference counting
      - Keep a reference counter as part of each object. Decrement the counter every time a reference to the object goes out of scope, and increment it every time a new reference to the same object is made. When the counter goes to 0, delete the object.
      - Adv: Fast and with very little runtime overhead
      - Dis: Cannot handle cyclic references
        - see http://www.cs.bilkent.edu.tr/~bgedik/coursewiki/doku.php/cs315:refc
    - There are other algorithms that can handle all cases, but require a more complex runtime procedure to be run, which might cause slowness when it kicks in
      - One example is the Mark & Sweep algorithm
        - In the mark phase, start from root variables (variables that are on the stack and static variables) and trace them to find all accessible objects and mark them
        - In the sweep case, scan the heap, deallocating unmarked objects and unmarking the marked ones