

Types

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**

- C and C++ are **weakly typed**. C permits a wide range of really bad things to occur through both casting and the union data structure.
- Java are **strongly typed** and are designed to catch all type errors at compile or run time.
- The **benefit** of a **strongly typed** language is that the compiler or run-time engine tends to find type errors due to programmer logic, resulting in **more reliable programs**.
- The **drawback** of a strongly typed language is that it makes it **more difficult to implement certain algorithms** or **manipulate or view memory directly**.

Basic Types

- Most programming language **data types** are **built on a set of** more or less **standard types**, because the **computational hardware** is designed to **support them**.
 - The **instruction sets** of most CPUs **include functions for** handling **integer** data types and **floating point** data types.
- String as a type in different languages
 - C considers a **String as an array of characters plus a terminator**, NULL at the end. While a character in C is a char type value, which is interchangeable with the corresponding int value that is the ASCII value of the character.
 - Java treated **String as a class**. Even if we define String s = "Hello World", Java implicitly creates a String object for us using String s = new String("Hello World"). Java doesn't have the char type, so it considers a string as **an array of byte, and uses the platform's default character set for decoding** (ASCII or Unicode).
 - Strings in **Python** are **arrays of bytes representing unicode characters**. There is not data type representing a single character in Python. A single character in Python is a string with length 1.
- In almost any language, it is possible to build data types of arbitrary form and format, creating appropriate operators for each type. The specifics of the language determine the level of difficulty of implementing a specific type.

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**. However, since **Python** chose to **use the longest data types** for representing basic types, there is **less likelihood of a problem**.
 - Show typeFormats.c/java/python
 - The reason why the size is 24 in Python is because every Python object contains at least a refcount and a reference to the object's type in addition to other storage; on a 64-bit machine, that takes up 16 bytes. REF: <http://stackoverflow.com/questions/10365624/sys-getsizeofint-returns-an-unreasonably-large-value>
- 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 that set of implicit conversions for numeric types in expressions with binary operators. The idea is to keep the precision after conversion. (float $\rightarrow$ double, small size $\rightarrow$ large size)

```
/**
 * conversion loses precision
 */
#include <stdio.h>

int main () {
    float f = 4.3;
    int i = f;
    char c = i;

    printf("i = %d, c = %d\n", i, c);

    return 0;
}
```

```
/**
 * implicit conversion (overflow)
 * conversions..c
 * Ying Li
 * 10/01/2017
 */

#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;
}
```

- Java has much more complicate rules, since it needs to define the rules not only for primitive types but also non-primitive types, for example, the conversion between super type and subtype.
 - Show the ConversionI.java, ask what is the output of the program. [Run-time error]

```

/**
 * File: ConversionI.java
 * Author: Ying Li
 * Date: 09/29/2019
 */

public class ConversionI {

    public static void main (String[] argv) {
        Parent p1 = new Parent(100);
        System.out.println("p1 is " + p1);
        System.out.println("p1 is happy " + p1.isHappy());

        Child c1 = new Child(101);
        System.out.println("c1 is " + c1);
        System.out.println("c1 is happy " + c1.isHappy());

        Parent p2 = new Child(102);
        System.out.println("p2 is " + p2);
        System.out.println("p2 is happy " + p2.isHappy());

        Child c3 = (Child) p2;
        System.out.println("c3 is " + c3);
        System.out.println("c3 is happy " + c3.isHappy());

        Child c2 = (Child) p1;
        System.out.println("c2 is " + c2);
        System.out.println("c2 is happy " + c2.isHappy());
    }
}

class Parent {

    protected long id;

    public Parent (long id) { this.id = id; }

    public String toString () { return Long.toString(id); }

    public boolean isHappy () { return false; }
}

class Child extends Parent {

    public Child (Integer id) { super(id); }

    public boolean isHappy () { return true; }
}

```

- If change Child c2 = p1, what is the output? [Compiler-time error]

- Downcasting and upcasting in Java
 - Downcasting and upcasting are applied on objects.
 - Upcasting, cast subtypes to super types, implicit. `Parent p2 = (Parent) new Child(102);` `Parent p2 = new Child(102)`
 - Downcasting, cast super types to subtypes, explicit. May cause `java.lang.ClassCastException`. `Child c2 = (Child) p1`
- Can use `instanceof` to decide the type of instance before downcasting

```
Cat c1 = new Cat();
Animal a = c1;
//if (a instanceof Cat)
Cat c2 = (Cat) a;

Mammal m = new Mammal();
Cat c = (Cat) m; // run-time error java.lang.ClassCastException
```

Pointers

- What is a pointer?
 - Pointers are a **data type** that enables explicit memory addressing.
 - A pointer **holds the address** of a variable of a certain type.
 - A pointer **does not hold data**, although the address itself can be treated as data in some situations.
- Deference operator
 - The deference operator **applies only to pointers**
 - The deference operator **provides the value of the variable at the address** specified by the pointer
- Strongly typed languages vs. weakly typed languages
 - In a strongly typed language, the pointer must have the same base type as the variable it addresses
 - In a weakly typed language, the pointer does not have to be the same type as the variable it addresses
- Do all languages have pointers?
 - **All languages have pointers.**
 - **Not all languages call them pointers or give** the user **explicit operators** for managing them.
 - C/C++ give users the explicit operators for managing pointer. Star (*) is used to define a pointer, and ampersand (&) is used to dereference.
 - Java does not provide explicit operators. But all variables, except the variables of the primitive type are pointers. Instead of using the word pointer, **Java uses reference**, and references are **strongly typed**. **Dereferencing in Java is automatic.**

```
/**
 * All the variables involved are pointers,
 * but requires no explicit dereferencing
 */
public class deref {
    public static void main (String[] argv) {
        Integer a = new Integer(4);
        Integer b = new Integer(6);
        Integer c;

        c = a + b;
        System.out.println("c is: " + c);
    }
}
```

```
/**
 * Variables must be explicitly allocated and
 * explicitly freed. All dereference must be
 * explicit.
 */
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char* argv[]) {
    int *a = malloc(sizeof(int));
    int *b = malloc(sizeof(int));
    int *c = malloc(sizeof(int));

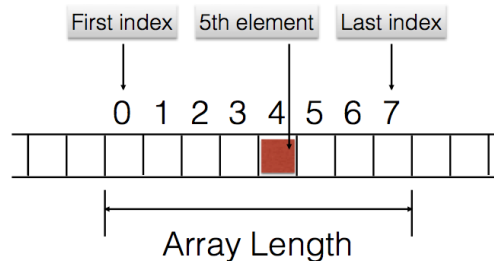
    *a = 4;
    *b = 6;
    *c = *a + *b;
    printf("c is %d\n", *c);

    free(a);
    free(b);
    free(c);

    return(0);
}
```

Arrays

- What is an array?
 - Arrays are indexed sequences of elements stored in chunks of continuous memory.
 - An array can be thought as a virtual grid of a particular type size laid over memory.



- Why use arrays? What are their advantages?
 - Efficient use of space and random access in constant time
- How can it access any location $A[i]$ in constant time?
 - On the hardware level, just need to find the address in memory of $A[i]$.
 - address of $A[i] = \text{address-of-}A + i * \text{slot-size}$
- Array in C and Pointer Arithmetic
 - Arrays and Pointers are the same thing in C [show pointers.c]
 - You allocate some memory and assign it to a pointer and that memory is automatically an array.
 - You can use array syntax or pointer syntax to access the slots in the array.

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main (int argc, char *argv[]) {
    int i;
    int *x = malloc(sizeof(int) * 6);

    x[0] = 1;
    *(x + 1) = 2;

    i = 2;
    x[i] = 3;

    i++;
    i[x] = 4;

    i++;
    *(x + i) = 5;

    //printf("addr(x): %p\n", x);
    x += 5;
    //printf("addr(x): %p\n", x);
    *x = 6;
    x -= 5;

    printf("%d %d %d %d %d %d\n", x[0], x[1], x[2], x[3], x[4], x[5]);

    free(x);

    return (0);
}
```

*The pointer's value is incremented/
decremented by the units of the type size*

*the size of an int is 4 bytes, then the
expression $x += 5$ increment x by 20*

Multidimensional Arrays

- How do you make it into a continuous block of memory?
 - by rows or by columns

```
A[0][0] A[0][1] A[0][2]
A[1][0] A[1][1] A[1][2]
A[2][0] A[2][1] A[2][2]
```

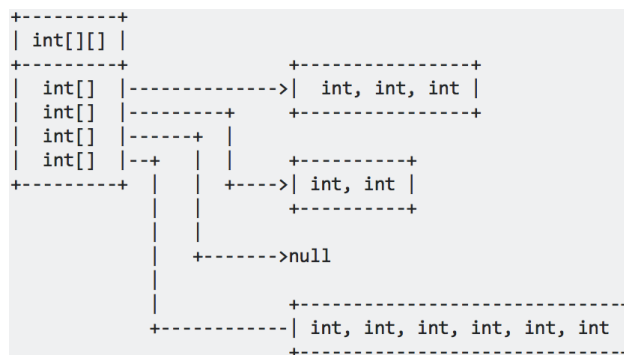
- row-major order

A[0][0]	A[0][1]	A[0][2]	A[1][0]	A[1][1]	A[1][2]
---------	---------	---------	---------	---------	---------

- column-major order

A[0][0]	A[1][0]	A[2][0]	A[0][1]	A[1][1]	A[2][1]
---------	---------	---------	---------	---------	---------

- Again, we need random access in constant time. How do we compute the address of $A[i][j]$?
 - In row major order, it is
 $\text{addr of } A[i][j] = \text{addr-of-A} + i * \text{slot-size} * \text{row-size} + j * \text{slot-size}$
- Does row-major vs col-major make any difference? If you are searching the array which order should you do it? Why?
 - same as hardware; cache hits increase
- Which way does Java do it?
 - Neither. Advantage of Java way is you can have irregular shaped arrays.
 - Java has arrays of arrays. While a given array's entries are stored in a contiguous block memory, the subordinate arrays those entries point to are object references to complete separate unrelated blocks of memory.



So $a[2][3]$ means “Get the array referenced by the entry at index 2 of a , then get the entry referenced by index 3 of that subordinate array.”

Type of Data in Arrays

- In statically typed languages, need all entries to be the same type. Why?
 - To avoid having to type cast and so that all slots have same size so can calculate the index.
- Then how do dynamically typed languages handle arrays? Since you can put any size of data in each slot, how can the interpreter or compiler calculate $A[i]$ efficiently?
 - Each slot contains pointers to the data and all pointers are the same size.

Dope Vectors

- What if you want to pass a 2D array pointer as a parameter to a procedure? How does the receiving procedure know how many rows and columns there are? Without knowing that, the procedure can't figure out where $A[i][j]$ is located.
- Solution: In Java, an array is an object and one of the fields of the object is its length. In many other languages, the information is provided in a *dope vector*, which is a block of data containing information about the array, including the number of rows and columns, starting and ending indices if not 0 and $n-1$, size of each slot, etc.
- The implementation of dope vectors can vary from language to language. When you are passed a pointer to an array, it might be that you are passed actually a pointer to the dope vector, which might include a pointer to the array. Or you are passed a pointer to the array and the slot at index-1 contains a pointer to the dope vector. Or the dope vector might just reside immediately in front of the array.

First Element Index

- In C/C++/Java and most modern languages, arrays are indexed starting from 0.
- But other languages, like Matlab and Fortran, typically start at index 1.
- Other languages like Pascal, allow you to specify any starting and ending index, so your array could go from index -5 to +5 and have 11 slots in it.
- Does it matter? In C it does, since arrays and pointers are identical, but not in other languages. I think 0.5 should be used as the starting index as a compromise between 0 and 1. =>

String as a Data Type

- What is a string?
 - Array of characters
- Built-in type in most languages so you don't need to think of it as an array. (Java, Python)

```
String greeting = "Hello";
```

- In C, it is a null-terminated array of ASCII characters

```
char[] greeting = "Hello";  
char[6] greeting = {'H', 'e', 'l', 'l', 'o', '\0'};
```

Structures

- Like objects in OO languages with no methods and with public fields
- Access the data of a structure uses the structure reference . usually. But, if the structure variable is a pointer, use -> to access the data. [show student.c student2.c]
- One issue with structures is that the modern design of computer buses and caching works best if memory accesses are on N-byte orders (e.g., N = 4). A structure with a short and an int in it might have a size of 8 bytes, as would a structure with two shorts and an int. On the other hand, a struct with three chars in it has a size three. The exact rules may be implementation and architecture dependent.

Functions as data types

- Functions can also be a data type that can be passed as parameters and assigned to variables
- Why would you want to pass a function as parameters?
 - to perform operations like comparing or filtering or combining
- Example - Python (show functions.py, ask the outputs)

```
def foo (x):
    print 3+x

bar = foo
bar(5) #prints 8

def baz(fn):
    fn(6)

baz(bar) #prints 9
```

- Easy with Python, but for statically typed languages, you have to declare the type of the function. How do you specify the function type?

```
#include <stdio.h>

void print_int (int i) {
    printf("%d\n", i);
}

int main() {
    // declare a function pointer
    void (*func)(int);

    // initialize a function pointer
    func = print_int;

    // call a function pointer
    func(10);

    return 0;
}
```

- Example - C (show functionPointer.c)
- Example - C qsort (show qsortExample.c)
 - #include <stdlib>

```
/**
 * Example of qsort: sort parts of a drone in ascending order in terms of price
 * Ying Li
 * 10/03/2019
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
    char name[50];
    int price;
} part;

int compare (const void* p1, const void* p2) {
    int m = ((part *)p1)->price;
    int n = ((part *)p2)->price;
    return m-n;
}

int main () {
    part drone[6];

    strcpy(drone[0].name, "DJI Flamewheel Kit");
    drone[0].price = 299;

    strcpy(drone[1].name, "Navio2");
    drone[1].price = 117;

    strcpy(drone[2].name, "Power Module");
    drone[2].price = 22;

    strcpy(drone[3].name, "Receiver");
    drone[3].price = 30;

    strcpy(drone[4].name, "Batteries");
    drone[4].price = 60;

    strcpy(drone[5].name, "Sonar");
    drone[5].price = 25;

    qsort (drone, 6, sizeof(part), compare);

    for (int i = 0; i < 6; i++) {
        printf("%s: %d\n\n", drone[i].name, drone[i].price);
    }

    return 0;
}
```

- `void qsort(void *base, size_t nmem, size_t size, int(*compar)(const void *, const void *));`
- return value of compar function
 - < 0 the first parameter is smaller than the second one
 - = 0 two parameters are the same
 - > 0 the first parameter is larger than the second one

Polymorphism

- A function is polymorphic if ???
 - It can be applied to several different types of data
- Why function polymorphism?
 - Avoid code duplication
- How is it implemented?
 - Depends
 - C uses the void * data type and function pointers to provide polymorphism (qsort is an example)
 - Java uses generic type (Stack.java)

```

"""
Generic Stack
Ying Li
03/08/20
"""

class Stack:

    def __init__(self):
        self.stack = []

    def push(self, item):
        self.stack.append(item)

    def pop(self):
        if not self.isEmpty():
            return self.stack.pop()
        else:
            return None

    def peek(self):
        if not self.isEmpty():
            return self.stack[len(self.stack)-1]
        else:
            return None

    def isEmpty(self):
        return len(self.stack) == 0

def main():
    stk = Stack()

    print(stk.pop())

    for i in range(0, 5):
        stk.push(i)

    print("top is: ", stk.peek())

    for i in range(0, 5):
        print("pop: ", stk.pop())

    print(stk.peek())

if __name__ == '__main__':
    main()

```

```
/**
 * Generic Stack
 *
 * Ying Li
 * 03/09/2020
 */
class Stack<T>{
    Node<T> top;

    T pop () {
        if (top != null) {
            T item = top.data;
            top = top.next;
            return item;
        }
        return null;
    }

    void push (T item) {
        Node<T> t = new Node<T> (item);
        t.next = top;
        top = t;
    }

    T peek () {
        return top.data;
    }

    boolean isEmpty () {
        if (top == null) {
            return true;
        } else {
            return false;
        }
    }

    public static void main (String[] args) {
        Stack<Integer> s = new Stack<Integer>();
        s.push(3);
        s.push(4);
        s.push(5);
        while (!s.isEmpty()) {
            System.out.println(s.pop());
        }
    }
}

class Node<T> {
    Node<T> next = null;
    T data;

    public Node (T d) {
        data = d;
    }
}
```

- Python is a dynamically typed language, it supports different data types naturally (Stack.py)
- Polymorphism in OO languages?
 - Inheritance
 - A class inherits characteristics from parent classes
 - Override
 - A subclass redefine the method inherited from a super class
 - Overriding vs overloading
 - Overriding: same method name, same parameters in different class (run-time concept)
 - Overloading: same method name, different parameters in one class (compile-time concept)
 - Abstract class
 - Can be inherited only, cannot have instances
 - Java: declared by using the keyword, abstract
 - Python: a class with at least one abstract method
 - ✓ abstract method: a method that has declaration but not implementation
- Multiple inheritance:
 - inherit more than one class (Python class can inherit from multiple classes)

```

"""
Conversion.py
Ying Li
03/08/2020
"""

class Human:

    def setID (self, id):
        self.id = id

    # abstract method
    def isHappy (self):
        pass

    def printt (self):
        print(self.id, ", ", self.isHappy())

class Parent (Human):

    def __init__ (self, id):
        Human.setID(self, id)

    def isHappy (self):
        return False

    # python doesn't support overloading
    def isHappy (self, age):
        if age < 25:
            return True
        else:
            return False

    # isHappy invoked in printt is the most recent one
    # so the interpreter complains it doesn't have param
    def printt (self):
        print("ID %d is %s" % (self.id, self.isHappy()))

```

```

class Child (Human):

    def __init__ (self, id):
        Human.setID(self, id)

    def isHappy (self):
        return True

def main ():
    p = Parent(100)
    c = Child(101)

    p.printt()
    c.printt()

if __name__ == "__main__":
    main()

```

```
/**
 * File: ConversionII.java
 * Author: Ying Li
 * Date: 03/08/2020
 */

public class ConversionII {

    public static void main (String[] argv) {
        Parent p = new Parent(100);
        Child c = new Child(101);
        Human h1 = p;
        Human h2 = c;

        h1.print();
        h2.print();
    }
}

abstract class Human {
    protected int id;

    public void setID (int id) { this.id = id; }
    public abstract boolean isHappy ();
    public void print () { System.out.println(id + ", " + isHappy()); }
}

class Parent extends Human {

    public Parent (int id) {
        setID (id);
    }

    public boolean isHappy () { return false; }

    // overloading
    public boolean isHappy (int age) {
        if (age < 25) { return true; }
        else { return false; }
    }

    // overriding
    public void print () { System.out.printf("ID %d is %s\n", id, isHappy()); }
}

class Child extends Human {

    public Child (int id) {
        setID (id);
    }

    public boolean isHappy () { return true; }
}
```

- Java can allow one subclass extend from one super class, but it allows a class implement multiple interfaces.