Basic Types

*Programming Languages*

*William Killian*

Millersville University
Outline

• Primitive Data Types
• Pointers and References
• Sum Types
  • Enumerations
  • Optional
  • Expected
  • Variants
• Product Types
  • Records
  • Tuples
• Strings?
Primitive Data Types
Primitive Data Types

• The fundamental building-blocks of programming
• Three main categories
  • boolean
  • integral
  • floating-point
• What makes them “primitive”?
  • Stored directly as-is in memory
  • Bit-for-bit stored in registers
    • a special (super fast) memory storage location in hardware
  • Supported operations are implemented in hardware
**boolean**

- Domain of values:
  - `true`
  - `false`

- Representation:
  - Representing a single-bit in hardware is often impossible
  - Instead, use a single byte (8 bits)
  - Language-dependent:
    - C/C++: “all zeroes” denotes `false`, anything else denotes `true`
    - Java: must explicitly use `true` and `false`
integral

- Numerical values represented in a power-of-two notation. Possible implementations:
  - unsigned \( 2^{n-1} 2^{n-2} \ldots 2^2 2^1 2^0 \)
  - one’s complement \( -(2^n - 1) + (2^{n-1} 2^{n-2} \ldots 2^2 2^1 2^0) \)
  - two’s complement \( -2^{n-1} 2^{n-2} \ldots 2^2 2^1 2^0 \)

- Bit: binary digit

- 8-bit integral numbers contain 8 individual bits which can have any permutation of values
integral

• Common sizes:
  • 8-bit (char)
  • 32-bit (int)
  • 16-bit (short)
  • 64-bit (long)

• Common language implementations
  int
  long
  Int/Long
  i32/u64

Python, C, C++, Java, OCaml, Ruby
C, C++, Java
Swift
Rust
floating-point

• Numbers that have a decimal point
• Often some advanced hardware-based representation (e.g. IEEE 754)
• Various sizes (32, 64) change range and precision
• Common Language Implementations
  • float  Python, C, C++, Java, Ocaml, Ruby
  • double  C, C++, Java (larger)
  • Float/Double  Swift
  • number  TypeScript
  • f32/f64  Rust
Pointers and References
Pointers and References

• Some Programming Languages provide a direct abstraction to a memory model

• Pointer
  • “points” to a memory location
  • Abstraction: memory is just a large array of bytes
  • Interpret what is at that location as a specific type

• Reference
  • “refers” to a pre-existing entity
  • Usually called an alias (alternative name)

Most “newer” languages hide pointers
• Point to a location in memory (or null)
• Accessing null or an invalid memory location: BIG PROBLEM
• Languages with Pointers:
  • C/C++
  • BASIC
  • FORTRAN
  • COBOL
  • Go
  • OCaml
• Languages with “Hidden” Pointers:
  • Java
  • Ruby
• Refer to an existing entity
• Solves the “deference” pointer issue with `null`
• Languages with References:
  • C++
  • Swift
  • Rust
Case Study: C++

- Pointer types get * added
- Reference types get & added

- To Reference from Pointer:
  ```
  auto& ref = *ptr;
  ```

- To Pointer from Reference:
  ```
  auto* ptr = &ref;
  ```
```cpp
int a = 4;
int& b = a;
b++; // value of a ?
int* p = &a;
int* q = p;
a = 7;
// value of p ?
// value of *p ?
```
Sum Types
Sum Types

- When we think of “sum” we think of **addition**
- All types have a possible range of values
  - boolean \{ true, false \}
  - uint \{ 0, 1, 2, ..., 4294967294, 4294967295 \}
- Sum types “add” the possible range of values together to the range of the new type

**Sum Types allow us to:**

- Combine pre-existing types and allow one to be “selected” at any given time
- Create new datatypes for “tagging” information
Basic
Sum
Types

• Enumerations
• Optional
• Expected
• Variant

When you hear **sum**
... think **or**
Enumerations

• Give us a way to specify non-integral values
• Often used to define a new class of information
• Examples:
  • **Months**: January, February, March, April, ...
  • **Card Suits**: Clubs, Diamonds, Hearts, Spades
  • What else?

```c
// c
enum suit {
    CLUBS, DIAMONDS, HEARTS, SPADES
};
```
Optional

• When we want a choice of Something or Nothing
• Two classes:
  • *Something* of some type we care about
  • Nothing (*None*)

// C++
std::optional<int> v; // initially nothing
v = 4;

(* Ocaml *)
let x : int option = Some 4
let y : int option = None
Expected

• Gives us a way to specify a return value or an error if something else happened
• Two Classes:
  • Result of some type we care about
  • Error of some error result we can inspect
• Similar in structure to Optional

```java
// some made up language
Expected<String, Error> data = load_file("big.txt")
if (data) { // valid
...
} else { // inspect error
  ... 
}
```
Variant

• When we want a choice with some possible set of values for each choice
• Optional and Expected are specific types of Variants
• Data can be tagged and can take on different forms

(* OCaml *)

```ocaml
type expr = Add of expr * expr
             | Mul of expr * expr
             | Var of string
             | Num of int

(* represents expr: x * (a + 4) *)

let e = Mul(Var("x"),Add(Var("a"),Int(4)))
```
Product Types
Product Types

• When we see “product” we think **multiplication**
• Product types **multiply** the range of possible values

**Using Product Types allows us to:**
• Aggregate (group) pieces of information together
• Create a new entity with named attributes
Basic Product Types

- Records
- Tuples

When you hear **product**
... think **and**
Records

• A group or collection of **named** entities
• Referred to as **classes** or **structs** in most languages
• Access data via name

```
// C++

```struct` Student {  // A student has
  `std::string` name;  // a name AND
  `int` id;  // an ID number AND
  `double` gpa;  // a GPA
};

Student s = {"Will", 327291, 3.38};  // fix student record
s.gpa = 4.0;  // fix student record
```
Tuples

- A group or collection of entities
- Access data via **location** (first, second, third, ...)

(* OCaml *)

```ocaml
let threeInts : int * int * int = (1, 2, 3)
let (first, _, _) = threeInts; (* get first *)
```

// C++

```cpp
std::tuple<int, int, int> threeInts {1, 2, 3};
int first = std::get<0>(threeInts);
```

# Python

```python
declare threeInts = (1, 2, 3)
first = threeInts[0]
```
Strings?
Strings

• A sequence of characters
• When “combined” can provide additional context and information

• Questions
  • Should we view strings as being a basic types?
  • Should we view strings as being a complex* type?
  • Or could it be both?

"Hello, World!" "bob" "racecar"
Primitive vs. Library Defined

• In some languages, Strings are primitive types
  • OCaml
  • JavaScript
  • Ruby
  • Python

• In other languages, they are not (library-defined)
  • C++
  • Swift
  • Rust
  • Java

• In other languages, they don’t exist**
  • C
Immutability

• **Mutable** means can be changed
• **Immutable** means cannot be changed

• Languages where strings are mutable:
  • Python
  • JavaScript
  • Rust (String)

• Languages where strings are immutable:
  • OCaml
  • Java
  • Rust (str)
Wrap Up
Conclusion

Primitives
• Values that can be directly implemented in hardware (memory)

Pointer and References
• Refer to existing instances of a particular type in memory
• Concept of a null memory address (pointer)

Sum Types
• Give us a choice between options (or)

Product Types
• Group types together (and); individually accessible

Strings?
• Can be primitive or not; can be mutable or not