Principles of Programming Languages

Type

Introduction

- What do you mean when you declare

```
int n;
```

  - Possible values of **`n`**?
  - Possible operations on **`n`**?

- How about **`freshman`** defined below?

```
typedef struct {
   char name[30];
   int student_number;
} Student;
Student freshman = {"John", 644101};
```

Data Type

- A data type is a set
  - When you declare that a variable has a certain type, you are saying that
    (1) the values that the variable can have are elements of a certain set, and
    (2) there is a collection of operations that can be applied to those values
- Fundamental issues for PL designers:
  - How to define a sufficient set of data types?
  - What operations are defined and how are they specified for a data type?

Evolution of Data Types

- Earlier PL's tried to include many data types to support a wide variety of applications, e.g. PL/I
- Wisdom from ALGOL 68:
  - A few basic types with a few defining operators for users to define their own according to their needs
- From user-defined type to abstract data type
  - The interface of a type (visible to user) is separated from the representation and set of operations on values of that type (invisible to user)

Uses for Types

- Program organization and documentation
  - Separate types for separate concepts
    - Represent concepts from problem domain
  - Indicate intended use of declared identifiers
    - Types can be checked, unlike program comments
- Identify and prevent errors
  - Compile-time or run-time checking can prevent meaningless computations, e.g., `3+TRUE–"Bill"`
- Support optimization
  - Example: short integers require fewer bits
  - Access record component by known offset

Overview

- Various Data Types (Sec. 6.2~6.9)
  - Primitive Data Types, Character String Types, User-Defined Ordinal Types, Array Types, Associative Arrays, Record Types, Union Types, Pointer and Reference Types
- Type Binding (Sec. 5.4.2)
- Type Checking (Sec. 6.10)
- Strong Typing (Sec. 6.11)
- Type Equivalence (Sec. 6.12)
- Theory and Data Types (Sec. 6.13)

Primitive Data Types

- Almost all programming languages provide a set of primitive data types
  - Those not defined in terms of other data types
  - Integer, floating-point, Boolean, character
- Some primitive data types are merely reflections of the hardware
- Others require only a little non-hardware support for their implementation

Character String Types

- Values are sequences of characters
- Design issues:
  - Is it a primitive type or just a special kind of array?
  - Should the length of strings be static or dynamic?
  - What kinds of string operations are allowed?
    - Assignment and copying: what if have diff. lengths?
    - Comparison (=, >, etc.)
    - Catenation
    - Substring reference: reference to a substring
    - Pattern matching

Character String in C

- C and C++
  - Not primitive; use `char` arrays, terminate with a null character (0)
  - A library of functions to provide operations instead of primitive operators
  - Problems with C string library:
    - Functions in library do not guard against overflowing the destination, e.g., `strcpy(src, dest);` What if `src` is 50 bytes and `dest` is 20 bytes?
- Java
  - Primitive via the `String` and `StringBuffer` class

Character String Length Options

- Static: COBOL, Java's `String` class
  - Length is set when string is created
- Limited dynamic length: C and C++
  - Length varying, up to a fixed maximum
  - In C-based language, a special character is used to indicate the end of a string's characters, rather than maintaining the length
- Dynamic (no maximum): SNOBOL4, Perl, JavaScript
- Ada supports all three string length options

aracter String Implementation

- Static length: compile-time descriptor
- Limited dynamic length: may need a run-time descriptor for length (but not in C and C++)
- Dynamic length: need run-time descriptor; allocation/deallocation is the biggest implementation problem

Collection of variable's attr.

Static string
Length
Address

Limited dynamic string
Maximum length
Current length
Address

User-Defined Ordinal Types

- Ordinal type: range of possible values can be easily associated with positive integers
- Two common user-defined ordinal types
  - **Enumeration**: All possible values, which are named constants, are provided or enumerated in the definition, e.g., C# example
    `enum days {mon, tue, wed, thu, fri, sat, sun};`
  - **Subrange**: An ordered contiguous subsequence of an ordinal type, e.g., `12..18` is a subrange of integer type

Enumeration Types

- A common representation is to treat the values of an enumeration as small integers
  - May even be exposed to the programmer, as is in C:

```
enum coin {penny=1, nickel=5, dime=10, quarter=25};
enum escapes {BELL='\a', BACKSPACE='\b', TAB='\t',
             NEWLINE='\n', VTAB='\v', RETURN='\r' };
```

- If integer nature of representation is exposed, may allow some or all integer operations:

Pascal: `for C := red to blue do P(C)`

C: `int x = penny + nickel + dime;`

Evaluation of Enumerated Type

- Aid to readability, e.g., no need to code a color as a number
- Aid to reliability, e.g., compiler can check:
  - Operations (don't allow colors to be added)
  - No enumeration variable can be assigned a value outside its defined range
  - Ada, C#, and Java 5.0 provide better support for enumeration than C++ because enumeration type variables in these languages are not coerced into integer types

Subrange Types

- Ada's design
  - Not new type, but rename of constrained versions

```
type Days is (mon, tue, wed, thu, fri,
  sat, sun);

subtype Weekdays is Days range
  mon..fri;

subtype Index is Integer range 1..100;

Day1: Days;

Day2: Weekday;

Day2 := Day1;
```

Subrange Types

- Usually, we just use the same representation for the subtype as for the supertype
  - May be with code inserted (by the compiler) to restrict assignments to subrange variables
- Subrange evaluation
  - Aid to readability: make it clear to the readers that variables of subrange can store only certain range of values
  - Reliability: assigning a value to a subrange variable that is outside specified range is detected as an error

Array Types

- Array: an aggregate of homogeneous data elements, in which an individual element is identified by its position in the aggregate
  - Array indexing (or subscripting): a mapping from indices to elements
  - Two types in arrays: element type, index type
- Array index types:
  - FORTRAN, C: integer only
  - Pascal: any ordinal type (integer, Boolean, char)
  - Java: integer types only
  - C, C++, Perl, and Fortran do not specify range checking, while Java, ML, C# do

Categories of Arrays

- Static: subscript ranges are statically bound and storage allocation is static (before run-time)
  - Advantage: efficiency (no dynamic allocation)
  - C and C++ arrays that include `static` modifier
- Fixed stack-dynamic: subscript ranges are statically bound, but the allocation is done at declaration time during execution
  - Advantage: space efficiency
  - C and C++ arrays without `static` modifier

Categories of Arrays

- Stack-dynamic: subscript ranges and storage allocation are dynamically bound at elaboration time and remain fixed during variable lifetime
  - Advantage: flexibility (the size of an array need not be known until the array is to be used), e.g., Ada

```
Get(List_Len); // input array size
declare
List: array (1..List_Len) of Integer;
begin
   ...
End // List array deallocated
```

Categories of Arrays

- Fixed heap-dynamic: storage binding is dynamic but fixed after allocation, allocated from heap
  - C and C++ through `malloc`
- Heap-dynamic: binding of subscript ranges and storage is dynamic and can change
  - Advantage: flexibility (arrays can grow or shrink during execution), e.g., Perl, JavaScript
  - C#: through `ArrayList;` objects created without element and added later with `Add`

```
ArrayList intList = new ArrayList();
intList.Add(nextOne);
```

Array Initialization and Operations

- Some languages allow initialization at the time of storage allocation
  - C, C++, Java, C# example

```
int list [] = {4, 5, 7, 83}
```

  - Java initialization of String objects

```
String[] names = {"Bob", "Jake",
  "Joe"};
```

- Ada allows array assignment and catenation
- Fortran provides elemental operations, e.g.,
  - + operator between two arrays results in an array of the sums of the element pairs of the two arrays

Associative Arrays

- An unordered collection of data elements indexed by an equal number of values (keys)
  - User defined keys must be stored
- Perl:

```
%hi_temps = ("Mon" => 77, "Tue" =>
 79, "Wed" => 65, …);
```

```
$hi_temps{"Wed"} = 83;
```

  - `$` begins the name of a scalar variable

```
delete $hi_temps{"Tue"};
```

  - Elements can be removed with `delete`

Record Types

- A **record** is a possibly heterogeneous aggregate of data elements in which the individual elements are identified by names
- COBOL uses level numbers to show nested records (others use recursive definition)

```
01 EMP-REC.
   02 EMP-NAME.
      05 FIRST PIC X(20).
      05 MID   PIC X(10).
      05 LAST  PIC X(20).
   02 HOURLY-RATE PIC 99V99.
```

References and Operations

1. COBOL
   - field_name OF record_name_1 OF ... OF record_name_n
2. Others (dot notation)
   - record_name_1.record_name_2. ... record_name_n.field_name

- Operations:
  - Assignment is common if types are identical
  - Ada allows record comparison
  - COBOL provides `MOVE CORRESPONDING`
    - Copies a field of the source record to the corresponding field in the target record

Unions Types

- A union is a type whose variables are allowed to store different types of values at different times

```
union time {
  long simpleDate;
  double perciseDate;} mytime;

...

printTime(mytime.perciseDate);
```

- Design issues
  - Should type checking be required?
  - Should unions be embedded in records?

Discriminated vs. Free Unions

- In Fortran, C, and C++, no language-supported type checking for union, called free union
- Most common way of remembering what is in a union is to embed it in a structure

```
struct var_type {
  int type_in_union;
  union {
     float un_float;
     int un_int; } vt_un;
} var_type;
```

Discriminated vs. Free Unions

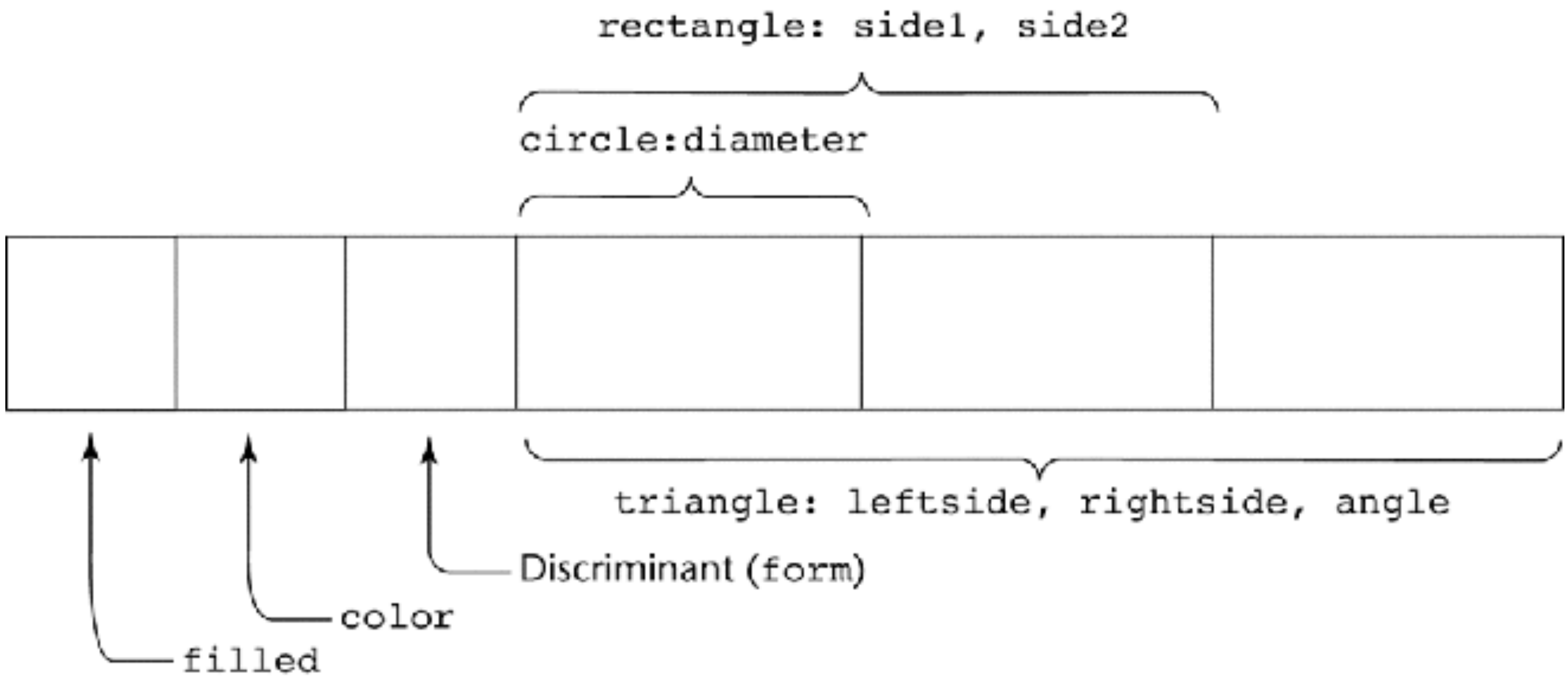
- Discriminated union: in order to type-checking unions, each union includes a type indicator called a discriminant
  - Supported by Ada
  - Can be changed only by assigning entire record, including discriminant → no inconsistent records

Ada Union Types

```
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 =>
  end case;
end record;
```

```
Figure_1 := (Filled => True,
Color => Blue, Form => Rectangle,
Side_1 => 2, Sice_2 => 3);
```

Ada Union Types



Evaluation of Unions

- Free unions are unsafe
  - Do not allow type checking
- Java and C# do not support unions
  - Reflective of growing concerns for safety in programming language
- Ada's descriminated unions are safe

Pointer and Reference Types

- A pointer type variable has a range of values that consists of memory addresses and a special value, nil
- Provide the power of indirect addressing
- Provide a way to manage dynamic memory
- A pointer can be used to access a location in the area where storage is dynamically created (heap)

Design Issues of Pointers

- What are the scope of and lifetime of a pointer variable?
- What is the lifetime of a heap-dynamic variable?
- Are pointers restricted as to the type of value to which they can point?
- Are pointers used for dynamic storage management, indirect addressing, or both?
- Should the language support pointer types, reference types, or both?

Pointer Operations

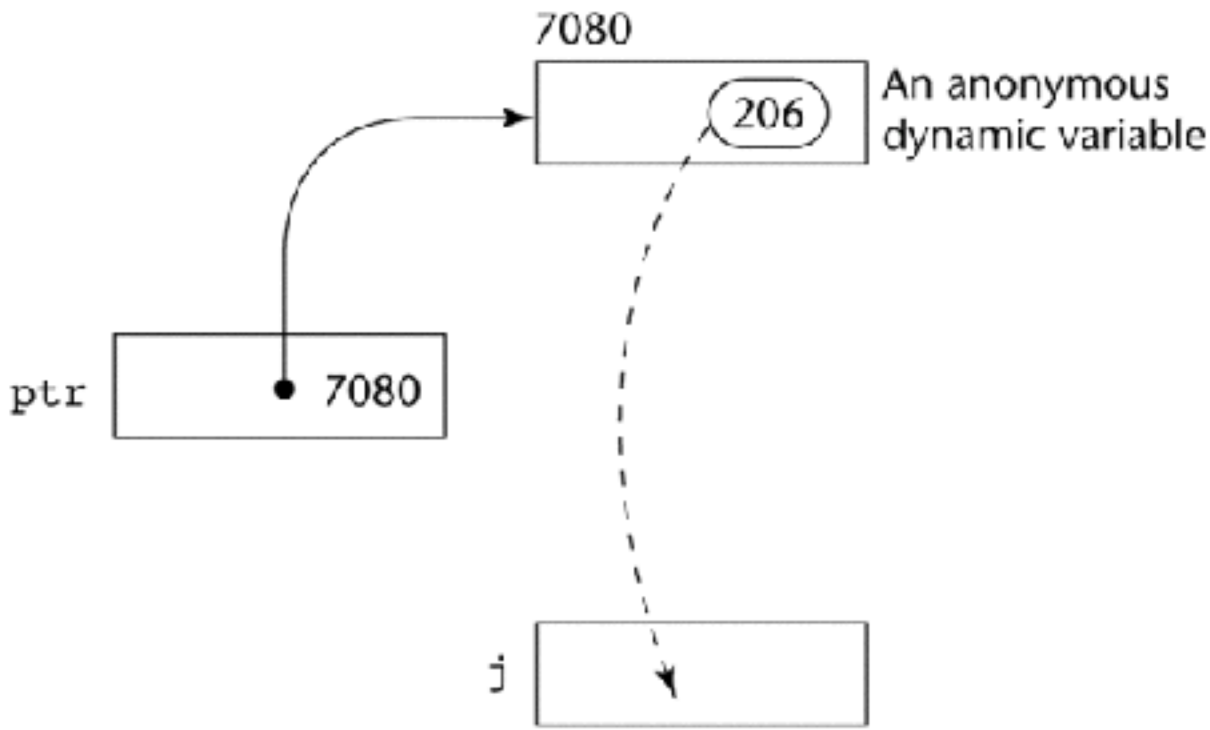
- Two fundamental operations: assignment and dereferencing
- Assignment sets a pointer variable's value to some useful address
- Dereferencing yields the value stored at the location represented by the pointer's value
  - Dereferencing can be explicit or implicit
  - C and C++ uses an explicit operator *

  ```
  j = *ptr
  ```

  sets `j` to the value located at `ptr`

Pointer Assignment

- -



Problems with Pointers

- Dangling pointers (dangerous)
  - A pointer points to a heap-dynamic variable that has been deallocated
    → has pointer but no storage
  - What happen when deferencing a dangling pointer?
- Lost heap-dynamic variable
  - An allocated heap-dynamic variable that is no longer accessible to the user program (often called garbage)
    → has storage but no pointer
    - The process of losing heap-dynamic variables is called memory leakage

Pointers in C and C++

- Extremely flexible but must be used with care
- Pointers can point at any variable regardless of when it was allocated
- Used for dynamic storage management and addressing
- Pointer arithmetic is possible
- Explicit dereferencing and address-of operators
- Domain type need not be fixed (`void *`)
  - `void *` can point to any type and can be type checked (cannot be de-referenced)

Pointer Arithmetic in C and C++

```
float stuff[100];
float *p;
p = stuff;
```

`*(p+5)` **is equivalent to** `stuff[5]` **and** `p[5]`

`*(p+i)` **is equivalent to** `stuff[i]` **and** `p[i]`

Reference Types

- A reference type variable refers to an object or a value in memory, while a pointer refers to an address
  → not sensible to do arithmetic on references

- C++ reference type variable: a constant pointer that is implicitly dereferenced; primarily for formal parameters → initialized with address at definition, remain constant

```
int result = 0;
int &ref_result = result;
ref_result = 100;
```

- Java uses reference variables to replace pointers entirely
  - Not constants, can be assigned; reference to class instances
    ```
    String str1;
    str1 = "This is a string.";
    ```

Evaluation of Pointers

- Dangling pointers and dangling objects are problems as is heap management
- Pointers are like `goto`'s--they widen the range of cells that can be accessed by a variable
- Pointers or references are necessary for dynamic data structures--so we can't design a language without them

Dangling Pointer Problem

- Tombstone: extra heap cell that is a pointer to the heap-dynamic variable
  - The actual pointer variable points only at tombstones
  - When heap-dynamic variable de-allocated, tombstone remains but is set to nil
  - Any pointer variables pointing to that heap-dynamic variable will know it is gone by noticing tombstone becomes nil
  - Costly in time and space

Dangling Pointer Problem

- Locks-and-keys:
  - Heap-dynamic variable represented as (variable, lock)
  - Associated pointer represented as (key, address)
  - When heap-dynamic variable allocated, a lock is placed in lock cell of that variable as well as the key cell of the corresponding pointer variable
- Any copies of the pointer value to other pointer variables must also copy the key value
- When a heap-dynamic variable is deallocated, its lock value is cleared to an nil
- Any remaining pointers will have a mismatch

Heap Management

- Two approaches to reclaim garbage
  - Reference counters (eager): reclamation is gradual
  - Garbage collection (lazy): reclamation occurs when the list of variable space becomes empty
- Reference counters:
  - A counter in every variable, storing number of pointers currently pointing at that variable
  - If counter becomes zero, variable becomes garbage and can be reclaimed
  - Disadvantages: space required, execution time required, complications for cells connected circularly

Garbage Collection

- Run-time system allocates cells as requested and disconnects pointers from cells as needed. Garbage collection when out of space
  - Every heap cell has a bit used by collection algorithm
  - All cells initially set to garbage
  - All pointers traced into heap, and reachable cells marked as not garbage
  - All garbage cells returned to list of available cells
  - Disadvantages: when you need it most, it works worst (takes most time when program needs most of cells in heap)

Overview

- Various Data Types (Sec. 6.2~6.9)
    - Primitive Data Types, Character String Types, User-Defined Ordinal Types, Array Types, Associative Arrays, Record Types, Union Types, Pointer and Reference Types
- Type Binding (Sec. 5.4.2)
- Type Checking (Sec. 6.10)
- Strong Typing (Sec. 6.11)
- Type Equivalence (Sec. 6.12)
- Theory and Data Types (Sec. 6.13)

Type Binding

- Before a variable can be referenced in a program, it must be bound to a data type
  - How is a type specified?
  - When does the binding take place?
- If static, the type may be specified by either
  - Explicit declaration: by using declaration statements
  - Implicit declaration: by a default mechanism, e.g., the first appearance of the variable in the program
  - Fortran, PL/I, BASIC, Perl have implicit declarations
    - Advantage: writability
    - Disadvantage: reliability (less trouble with Perl)

Dynamic Type Binding

- A variable is assigned a type when it is assigned a value in an assignment statement and is given the type of RHS, e.g., in JavaScript and PHP

```
list = [2, 4.33, 6, 8];
list = 17.3;
```

  - Advantage: flexibility (generic for processing data of any type, esp. any type of input data)
  - Disadvantages:
    - High cost (dynamic type checking and interpretation)
    - Less readable, difficult to detect type error by compiler
      → PL usually implemented in interpreters

Type Inference

- Types of expressions may be inferred from the context of the reference, e.g., in ML, Miranda, and Haskell

```
fun square(x) = x * x;
```

  - Arithmetic operator `*` sets function and parameters to be numeric, and by default to be **`int`**

```
square(2.75);  //error!
```

```
fun square(x) : real = x * x; //correct
```

Overview

- Various Data Types (Sec. 6.2~6.9)
  - Primitive Data Types, Character String Types, User-Defined Ordinal Types, Array Types, Associative Arrays, Record Types, Union Types, Pointer and Reference Types
- Type Binding (Sec. 5.4.2)
- Type Checking (Sec. 6.10)

Type Checking

- The activity of ensuring that the operands of an operator are of compatible types
  - A compatible type is one that is either legal for the operator, or is allowed to be implicitly converted, by compiler-generated code, to a legal type, e.g.,
    `(int) A =(int) B + (real) C`
  - This automatic conversion is called a coercion
- All type bindings static → nearly all type checking can be static
  Type binding dynamic → type checking dynamic

Strong Typing

- A programming language is strongly typed if type errors are always detected
  - Advantage: allows the detection of the misuses of variables that result in type errors
  - FORTRAN 77 is not: EQUIVALENCE
  - C and C++ are not: unions are not type checked
- Coercion rules can weaken strong typing
  - Example: `a` and `b` are **`int`**; `d` is **`float`**; no way to check `a + b` mistakenly typed as `a + d`

Type Equivalence

- Type checking checks compatibility of operand types for operators → compatibility rules
- Simple and rigid for predefined scalar types
- Complex for structured types, e.g., arrays, structures, user-defined types
  - They seldom coerced → no need to check compatibility
  - Important to check equivalence, i.e., compatibility without coercion → how to define type equivalence?

Name Type Equivalence

- Two variables have equivalent types if they are in either the same declaration or in declarations that use the same type name

- Easy to implement but highly restrictive:

  - Subranges of integer types are not equivalent with integer types, e.g., Ada

```
type Indextype is 1..100;
count : Integer; index : Indextype;
```

  - Formal parameters must be the same type as their corresponding actual parameters

Structure Type Equivalence

- Two variables have equivalent types if their types have identical structures
- More flexible, but harder to implement → need to compare entire structures
  - Are two record types compatible if they are structurally the same but use different field names?
  - Are two array types compatible if they are the same except the subscripts? e.g. [1..10] and [0..9]
  - Are two enumeration types compatible if their components are spelled differently?
  - How about type `celsius` & `fahrenheit` of `float`?

Type Equivalence in C

- Name type equivalence for struct, enum, union
  - A new type for each declaration not equivalence to any other type
- Structure type equivalence for other nonscalar types, e.g., array
- **`typedef`** only defines a new name for an existing type, not new type

Summary

- Data types of a language a large part determine that language's style and usefulness
- Primitive data types of most imperative lang. include numeric, character, and Boolean
- The user-defined enumeration and subrange types are convenient and add to the readability and reliability of programs
- Arrays and records included in most languages
- Pointers are used for addressing flexibility and to control dynamic storage management