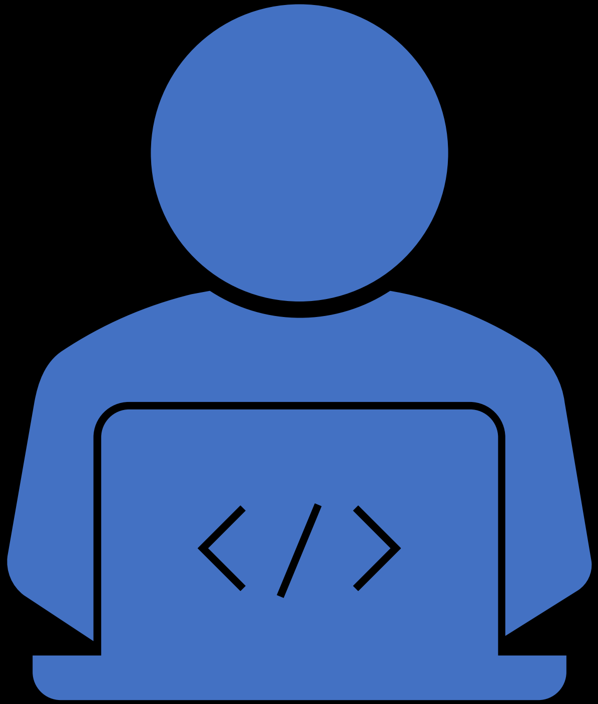


CENG204 - Programming Languages Concepts

Asst. Prof. Dr. Emre ŞATIR

Lecture 11
Data Types (Part 2)

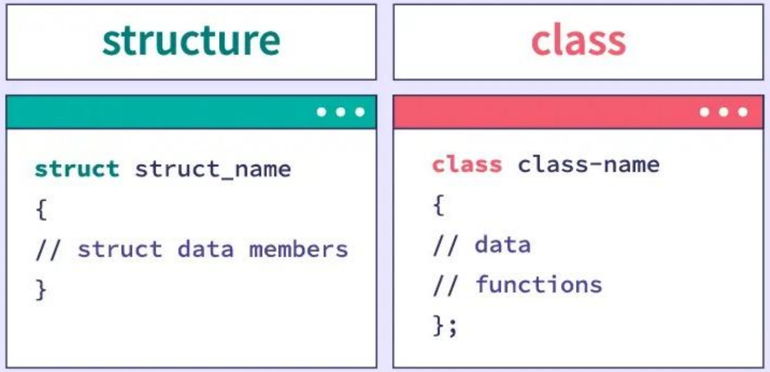
Lecture 11 Topics

- Record Types
- Tuple Types
- List Types
- Pointer and Reference Types
- Union Types
- Type Checking
- Strong Typing

Record Types

Record Types

- A **record** is a possibly heterogeneous aggregate of data elements in which the individual elements are identified by names.
- The fundamental difference between a record and an array is that record elements, or **fields**, are not referenced by indices. Instead, the fields are named with identifiers.
- The elements of a record are of potentially different sizes (different types) and reside in adjacent memory locations.
- In C, C++, and C#, records are supported with the **`struct`** data type.
- In Java, C++ and C#, records can be defined as data **classes**. Data members of such classes serve as the record fields.



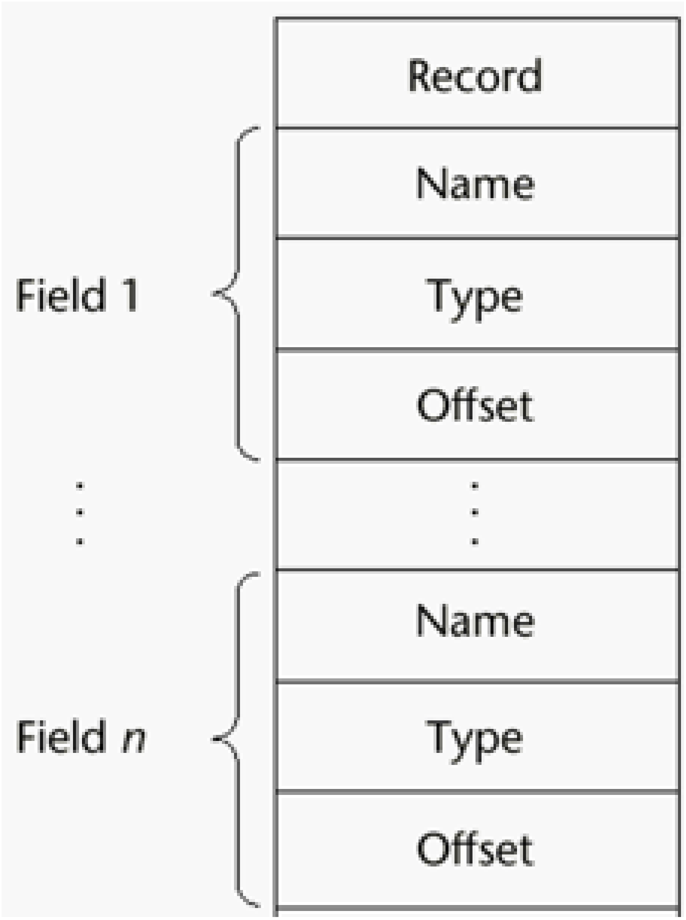
Implementation of Record Type

The fields of records are stored in adjacent memory locations.

But because the sizes of the fields are not necessarily the same, the access method used for arrays is not used for records.

Instead, the **offset address**, relative to the beginning of the record, is associated with each field.

Field accesses are all handled using these offsets.



References to Record Fields

- Most of the languages use "dot notation" for field references, where the components of the reference are connected with periods.
- They use the name of the largest enclosing record first and the field name last.
- For example, if `Middle` is a field in the `Employee_Name` record which is embedded in the `Employee_Record` record, it would be referenced with the following:

```
Employee_Record.Employee_Name.Middle
```

C Structures (Structs)

- **Structures** (also called **structs**) are a way to group several related variables into one place.
- Each variable in the structure is known as a **member** of the structure.

```
struct MyStructure
{
	int myNum;
	char myLetter;
};
```

C Structures (Structs)

```c
int main()
{
  struct myStructure s1;

  s1.myNum = 13;
  s1.myLetter = 'B';

  printf("My number: %d\n", s1.myNum);
  printf("My letter: %c\n", s1.myLetter);

  return 0;
}
```

Tuple Types

Tuple Types

- A **tuple** is a data type that is similar to a record, except that the elements are not named.
- The elements of a tuple need not be of the same type as records.
- Used in Python, ML, and F# to allow functions to return multiple values.
  - **Python**
    - Closely related to its lists, but immutable.
    - Is created by assigning a tuple literal, as in the following example:

      `myTuple = (3, 5.8)`

      Referenced with subscripts (begin at `0`)

      `myTuple[1]` → 5.8

Tuples in Python

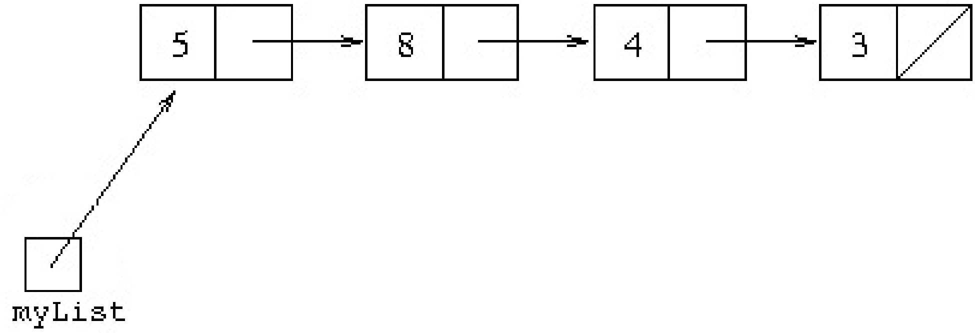
T = (20,   35.75)

T[0]   T[1]

- ✓ **Ordered**: Maintain the order of the data insertion.
- ✓ **Unchangeable**: Tuples are immutable and we can’t modify items.
- ✓ **Heterogeneous**: Tuples can contains data of types

List Types

List Types



- **Lists** were first supported in the first functional programming language, Lisp.
- Lists in Scheme and Common Lisp are delimited by parentheses and the elements are not separated by any punctuation. For example,

  `(A B C D)`

- Nested lists have the same form, so we could have

  `(A (B C) D)`

- In this list, `(B C)` is a list nested inside the outer list.

List Types - C# and Java

- They have always been part of the functional languages, but in recent years they have found their way into some imperative languages.
- Both C# and Java supports lists through their generic classes, `List` and `ArrayList/LinkedList`, respectively.

List Types - Python

- Python includes a list data type, which also serves as Python's arrays. Elements can be of any type.
- Unlike Python tuples, the lists of Python are mutable.
- A Python list is created with an assignment of a list value to a name. A list value is a sequence of expressions that are separated by commas and delimited with brackets.
- For example, consider the following statement:

```
myList = [3, 5.8]
```

- The elements of a list are referenced with subscripts in brackets, as in the following example:

`x = myList[1]` (This statement assigns 5.8 to x). The elements of a list are indexed starting at zero.

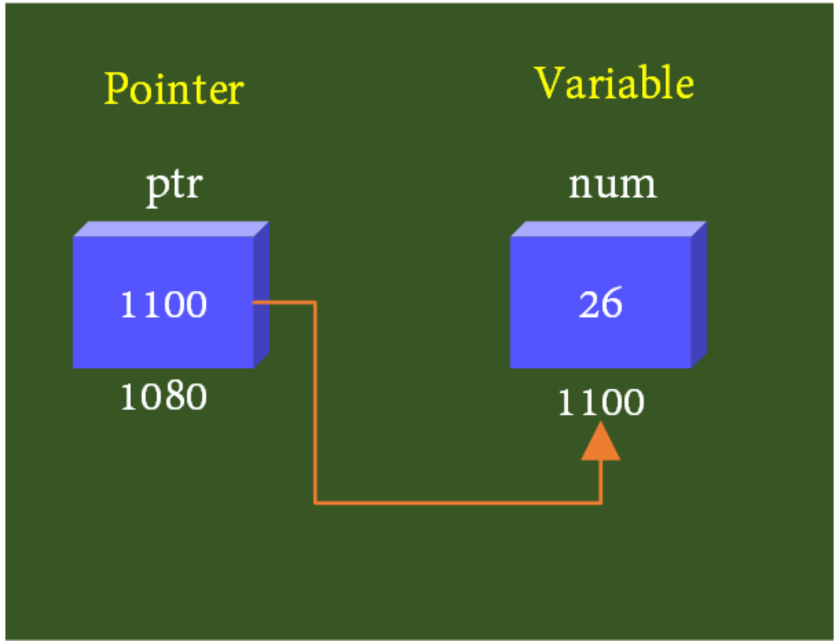
Differences between tuples and lists in python

list	tuple
1. list() is a collection of data that is ordered and changeable.	1. A tuple is collection of data that is ordered and unchangeable.
2. Python lists data are written in array brackets ex: []	2. Python tuples data are written in round brackets ex: ()

Pointer and Reference Types

Pointer and Reference Types

- A **pointer** type is one in which the variables have a range of values that consists of memory addresses and a special value, **nil** (or **null**).
- The value nil (null) is not a valid address and is used to indicate that a pointer cannot currently be used to reference a memory cell.
- Pointers are designed for two distinct kinds of uses:
  - First, pointers provide some of the power of "*indirect addressing*". (Indirect addressing allows the memory address to be varied so that it can point to more than one location at runtime).
  - Second, pointers provide a way to manage dynamic storage. A pointer can be used to access a location in an area where storage is dynamically allocated called a **heap**.



Pointer and Reference Types

- Variables that are dynamically allocated from the heap are called **heap-dynamic variables**.
- They often do not have identifiers associated with them and thus can be referenced only by pointer or reference type variables. Variables without names are called **anonymous variables**.
- "Reference variables", which are discussed later, are closely related to pointers.

Pointer and Reference Types

- Pointers add writability to a language.
- For example, suppose it is necessary to implement a dynamic structure like a binary tree in a language that does not have pointers or dynamic storage.
- This would require the programmer to provide and maintain a pool of available tree nodes, which would probably be implemented in parallel arrays.
- Also, it would be necessary for the programmer to guess the maximum number of required nodes.
- This is clearly an awkward and error-prone way to deal with binary trees.

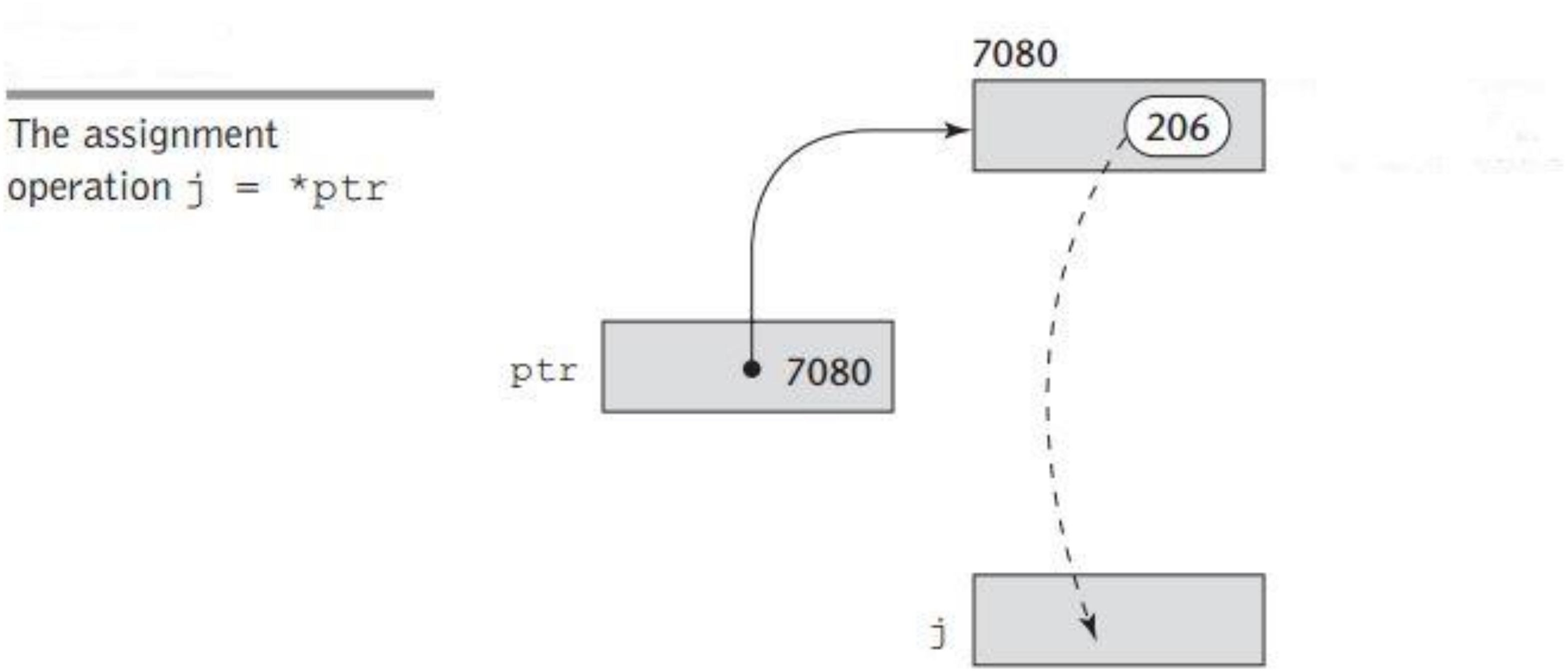
Pointer Operations

- Two fundamental operations: “assignment” and “dereferencing”.
- **Assignment** is used to set a pointer variable’s value to some useful address.
- **Dereferencing** yields the value stored at the location represented by the pointer’s value.
  - C and C++ uses an dereferencing operation via ‘*’ character.
  - For example;

    ```
    j = *ptr
    ```

    sets j to the value located at `ptr`.

Pointer Dereferencing Illustrated



Problems with Pointers

1. A **dangling pointer**, or dangling reference, is a pointer that contains the address of a heap-dynamic variable that has been deallocated.

- For example, in C++ we could have the following:

```
int * arrayPtr1;
int * arrayPtr2 = new int[100];
arrayPtr1 = arrayPtr2;
delete [] arrayPtr2;
```

Problems with Pointers

- In C++, both `arrayPtr1` and `arrayPtr2` are now dangling pointers, because the C++ delete operator has no effect on the value of its operand pointer.
- In C++, it is common (and safe) to follow a delete operator with an assignment of <u>zero</u>, which represents <u>null</u>, to the pointer whose pointed-to value has been deallocated.

- *** Notice that the <u>explicit deallocation</u> of dynamic variables is the cause of dangling pointers. Because Java class instances are “implicitly deallocated” (there is <u>no</u> explicit deallocation operator), there <u>cannot be dangling references in Java</u> (Instead of this Java has a “Garbage Collection” mechanism).

Problems with Pointers

2. **Lost heap-dynamic variable**
   - An allocated heap-dynamic variable that is no longer accessible to the user program (often called ***garbage***).
   - Lost heap-dynamic variables are most often created by the following sequence of operations:
     - Pointer `p1` is set to point to a newly created heap-dynamic variable.
     - Pointer `p1` is later set to point to another newly created heap-dynamic variable.
   - The process of losing heap-dynamic variables is called ***memory leakage***. Memory leakage is a problem, regardless of whether the language uses implicit or explicit deallocation.

Pointers in C and C++

- Extremely flexible but must be used with care.
  - This design offers no solutions to the dangling pointer or lost heap-dynamic variable problems.
- “Pointer arithmetic” is possible.
  - This feature makes their pointers more interesting than those of the other programming languages.

Pointers in C and C++

- In C and C++, the asterisk (*) denotes the <u>dereferencing</u> operation and the ampersand (&) denotes the operator for producing <u>the address of a variable</u>.
- For example, consider the following code:

```
int *ptr;
int count, init;
. . .
ptr = &init;
count = *ptr;
```

- The assignment to the variable `ptr` sets it to the address of `init`. The assignment to count dereferences `ptr` to produce the value at `init`, which is then assigned to `count`. So, the effect of the two assignment statements is to <u>assign the value of `init` to `count`</u>.

Pointer Arithmetic in C and C++

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

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

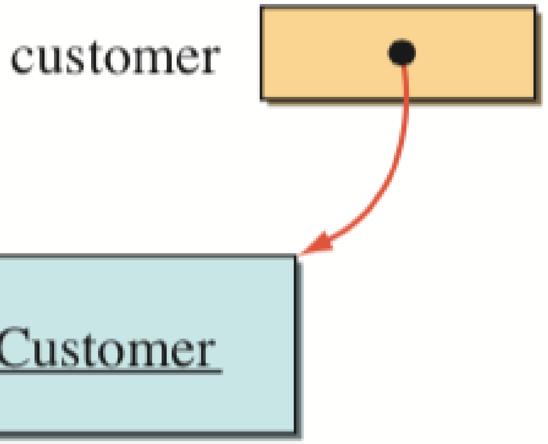
`*(ptr+i)` is equivalent to `stuff[i]` and `ptr[i]`

*** It is clear from these statements that the <u>pointer operations</u> include the same scaling that is used in <u>indexing operations</u>. Furthermore, pointers to arrays can be indexed as if they were array names.

Reference Types

- A **reference type variable** is similar to a pointer, with one important and fundamental difference: A pointer refers to an address in memory, while a reference refers to an object or a value in memory.
- As a result, although it is natural to perform arithmetic on addresses, it is not sensible to do arithmetic on references.
- All Java class instances (i.e., objects) are referenced by reference variables.

```
Customer customer;
customer = new Customer();
```



Reference Types

- In the following, `String` is a standard Java class:

```
String str1;
. . .
str1 = "This is a Java literal string";
```

- In this code, `str1` is defined to be a reference to a `String` class instance or object.
- It is initially set to `null`.
- The subsequent assignment sets `str1` to reference the String object, "`This is a Java literal string`".

Evaluation of Pointers and Reference Variables

- Pointers have been compared with the "`goto`". The goto statement <u>widens the range of statements</u> that can be executed next. Pointer variables <u>widen the range of memory cells</u> that can be referenced by a variable.
- Perhaps the most damning statement about pointers was made by Hoare (1973): "*Their introduction into high-level languages has been a step backward from which we may never recover.*"
- Pointers or references are necessary for dynamic data structures--so we can't design a language without them.

Union Types

Union Types

- A **union** is a type whose variables may store different type values at different times during program execution.
- The problem of type checking union types, which is discussed later, is their major design issue.
- C and C++ provide union constructs in which there is no language support for type checking; the union in these languages is called **free union**.
- Type checking of unions require that each union include a type indicator called a **discriminant** (Supported by ML, Haskell, and F#).

Example of a C Union

```c
#include <stdio.h>

union Data {
    int i;
    float f;
};
```

```c
int main() {
    union Data data;

    data.i = 10;
    printf("int: %d\n", data.i);

    data.f = 3.14;
    printf("float: %f\n", data.f);

    // The previous values are overwritten:
    printf("int (again): %d\n", data.i);

    return 0;
}
```

Example of a C Union

- In C, a union uses the same memory space for all its members.
- In this example, int and float are written to the same address.
- The last assigned value invalidates the previous values.

  - In the `union Data` structure, the same memory area is shared by `int` and `float`.
  - First, `data.i = 10` is assigned → then `data.f = 3.14` is assigned.
  - When `data.f` is assigned, the value stored in `data.i` is **overwritten**, i.e., corrupted.
  - If the program later tries to read `data.i` again, it will get an **invalid or meaningless value** because the currently valid content is of type `float`.

- Thanks to this feature, union structures are often used in situations where memory savings are required or in hardware-level programming to interpret different types on the same data.

🛑 Why is it called a "free union"?

This structure is referred to as a "**free union**" because:

- The compiler **does not track** which member is currently active.
- It does **not issue warnings or errors** if you access a member of the union that is not currently valid.
- This weakens **type safety** and increases the likelihood of runtime bugs.

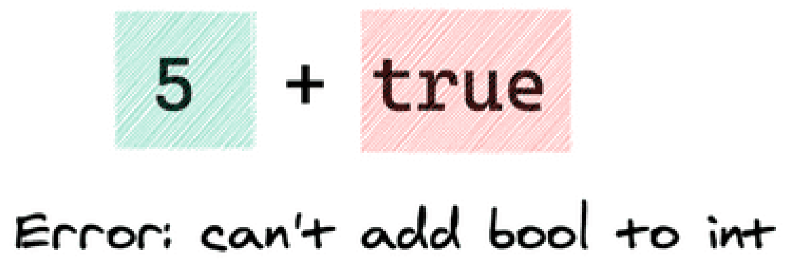
Evaluation of Unions

- Free unions are unsafe. They are one of the reasons why C and C++ are not **strongly typed**: These languages do not allow type checking of references to their unions.
- On the other hand, unions can be safely used, as in their design in ML, Haskell, and F#.
- Java and C# do not support unions.
  - Reflective of growing concerns for safety in programming language.

Type Checking

Type Checking

- **Type checking** is the activity of ensuring that the operands of an operator are of compatible types.
- A **compatible type** is one that either is legal for the operator or is allowed under language rules to be implicitly converted by compiler-generated code (or the interpreter) to a legal type.
- This automatic conversion is called a **coercion**.
- For example, if an int variable and a float variable are added in Java, the value of the int variable is coerced to float and a floating-point add is done.
- A **type error** is the application of an operator to an operand of an inappropriate type.



Type Checking

- If all bindings of variables to types are static in a language, then type checking can nearly always be done statically.
- Dynamic type binding requires type checking at run time, which is called **dynamic type checking**.
- Some languages, such as JavaScript and PHP, because of their dynamic type binding, allow only dynamic type checking.
- It is better to detect errors at compile time than at run time, because the earlier correction is usually less costly.
- The penalty for static checking is reduced programmer flexibility.

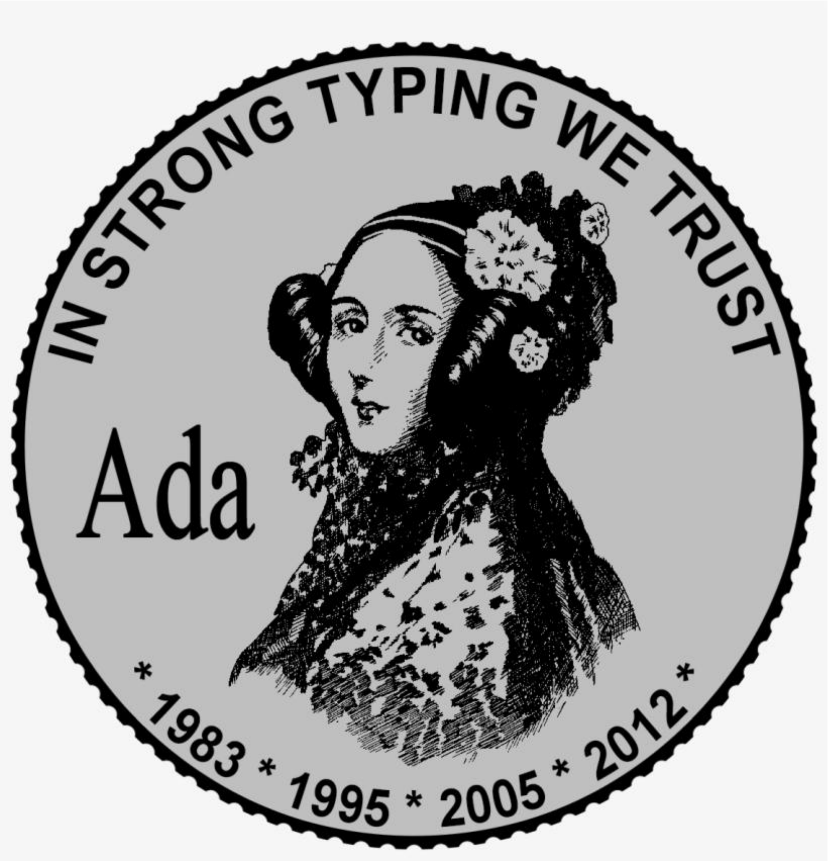
Strong Typing

Strong Typing

- One of the ideas in language design that became prominent in the so-called structured-programming revolution of the 1970s was **strong typing**.
- A programming language is **strongly typed** if <u>type errors are always detected</u>.
- This requires that the types of all operands can be determined, either at <u>compile time</u> or at <u>run time</u>.
- The importance and advantage of strong typing lies in its ability to <u>detect all misuses of variables</u> that result in type errors.

Strong Typing

- Ada, ML and F# are strongly typed languages.
- C and C++ are <u>not</u> strongly typed languages because both include union types, which are <u>not</u> type checked.
- Java and C#, although they are based on C++, are <u>nearly</u> strongly typed.
  - Types can be <u>explicitly cast</u>, which could result in a type error.



Strong Typing

- The coercion rules of a language have an important effect on the value of type checking.
- For example, expressions are strongly typed in Java.
- However, an arithmetic operator with one floating-point operand and one integer operand is legal.
- The value of the integer operand is coerced to floating-point, and a floating-point operation takes place.
- This is what is usually intended by the programmer.
- However, the coercion also results in a loss of one of the benefits of strong typing—error detection.

Strong Typing

- For example, suppose a program had the `int` variables `a` and `b` and the float variable `d`.
- Now, if a programmer meant to type `a+b`, but mistakenly typed `a+d`, the error would <u>not</u> be detected by the compiler.
- The value of `a` would simply be coerced to float.
- So, the value of strong typing is <u>weakened</u> by coercion.