CMSC 124

DESIGN AND IMPLEMENTATION OF PROGRAMMING LANGUAGES

DATA TYPES

DATA TYPE

Collection of data values and a set of predefined operations on those values.

Variable data types must match the real-world objects that they represent.

There are several reasons why a language must have a type system.

1.

Error detection, more specifically, type checking.

2.

Assists in enforcing program modularity; interface between program modules will be consistent.

3.

Documentation; variable data types give us information about the values they may hold.

Variables may be thought of as descriptors.

DESCRIPTORS

Collections of attributes that describe that variable, used for type checking and allocation and deallocation of variables.

These descriptors may either be needed at compile-time only (static variables), or maintained throughtout execution (dynamic variables).

Identifiers are not variables; they are just one of the attributes of variables.

Some important attributes of a data object are...

1.

TYPE

Describes the set of values the data object can take.

2.

LOCATION

The storage location (address) to which the data object is bound.

3.

VALUE

The value the data object holds; usually from an assignment statement.

4.

NAME

Also called the identifier; what the data object is called.

5.

COMPONENT

A data object may be part of other data objects, i.e., structures.

PRIMITIVE DATA TYPES

Data types that are not defined in terms of other types.

Instead, they are used to build structure types.

In general, the primitive data types are...

1.

Numeric types

1.1.

INTEGER

Represent mathematical integers usually as bits with a leftmost sign bit.

A language may have many integer types depending on size, i.e., `byte`, `int`, `short`, `long`, or the absence of the sign bit, i.e., `unsigned`.

The `unsigned` type is usually used for **binary data**.

Examples of languages that support `unsigned` types are C and C++; not all languages have an **`unsigned`** type.

Most integer types are hardware-supported.

However, some languages have integer types that are not hardware-supported, like Python, which has unlimited long integers:

Ex. `74985723495873452304578 2`

The most common binary representation of integers is 2's complement, because it is easier to store negative values and perform arithmetic (addition, subtraction).

Signed magnitude notation is longer used because it is not practical for arithmetic.

1's complement is no longer used because it has two representations for 0.

1.2.

FLOATING-POINT

Used to emulate real numbers.
Since they are limited by hardware, they are still only approximate.

Most irrational numbers and numbers that can not be represented in finite space can not be represented by floating-point values.

Floating-point types are still represented as binary.

The most common floating-point types are `float` and `double`.

`float` types use four bytes, `double` types use eight.

Because `double` types use twice as much space as float types, they are also called

DOUBLE-PRECISION TYPES.

PRECISION

Accuracy of the fractional part of the value.

1.3.

COMPLEX

Used to represent complex numbers, i.e., including imaginary numbers $\sqrt{-1}$.

Examples of programming languages that support complex types are FORTRAN and Python.

EXAMPLE: PYTHON

```
7 + 3j
```

The `j` stands for the imaginary number.

1.4.

DECIMAL

Numeric values specifically used for business applications; usually hardware-supported.

The number of decimal digits and the position of the decimal point are fixed in decimal types, unlike floating-point types.

EXAMPLE: COBOL

```
01 amount pic 999999V99.
```

Decimal point marked by **V**, two decimal places.

Binary Coded Decimal (BCD) representation is used to store decimal types.

2.

Boolean Type

BOOLEAN

Simplest data type; takes one of two values – `true` or `false`.

The primary use of `boolean` types in PLs is as switches or flags.

Some languages have a **separate boolean type** (e.g., Java, C#), others represent them as **numeric** (e.g., C, C++).

3.

Character types

CHARACTER

Although appearance is as symbols, representation in memory is still numeric (use of numeric codes).

Examples of character encoding/sets are ASCII and UNICODE.

Unicode was developed when ASCII (which could only represent 128 characters) was deemed inadequate.

The first language to utilize Unicode was Java.

Although characters are usually implemented as a primitive data type, some languages do not.

EXAMPLE: PYTHON

Python represents “characters” as strings with a length of 1.

```
var1 = 'a'
```

CHARACTER STRING TYPES

Have values that consist of sequences of characters.

Commonly used to label output.

Example:

```
void sum(int x, int y) {
     printf(“The sum of %d and %d
               is %d.\n”, x, y,
               (x+y));
}
```

Input and output of data is commonly in the form of strings.

They are also essential for programs that use character manipulation.

```
my $emailAddress = “”;
print “What is your Gmail address? “
$emailAddress = <STDIN>;
chomp($emailAddress);
while($emailAddress !~ /[a-zA-Z][a-zA-Z0-9]
                        @gmail\.com/) {
  print “You entered an invalid Gmail address.\n”;
  print “Please enter a valid Gmail address: “;
  $emailAddress = <STDIN>;
}
```

When implementing support for string types, there are a couple of design issues:

1.

Should strings be a special kind of character array or a primitive type?

2.

Should string length be static or dynamic?

Operations on strings include:

1.

ASSIGNMENT

Change the value of a string type variable by assigning a new string literal to it.

EXAMPLE: C

```
char name[30] = “Kei Peralta”;
```

2.

CONCATENATION

Joining two or more strings, end-to-end, usually as a binary infix operation.

EXAMPLE: C

```
#include<string.h>

...
char str1[20], str2[20],str3[40];

...
strcpy(str3, strcat(str1, str2));
```

EXAMPLE: JAVA

```
name = “Kei” + “ ” +
       “Peralta”;
```

EXAMPLE: JAVA

```
String s1, s2;

...

s1 = s1.concat(s2);
```

3.

SUBSTRING REFERENCE

Reference to a string within the current string, also called slices if represented as char arrays.

4.

LEXICAL COMPARISON

Determine if a string lexically greater than/less than /equal to another string.

5.

PATTERN MATCHING

Checks if a string follows a pattern, usually represented as a regular expression.

Examples of languages with built-in pattern matching support are Perl, Javascript, Ruby, and PHP.

On the other hand, C++, Java, Python, and C# have class libraries that support pattern matching.

A main issue in operations such as assignment and lexical comparison is when strings are of different length.

EXAMPLE

```
char str1[10], str2[30];
...
strcpy(str1, str2);
```

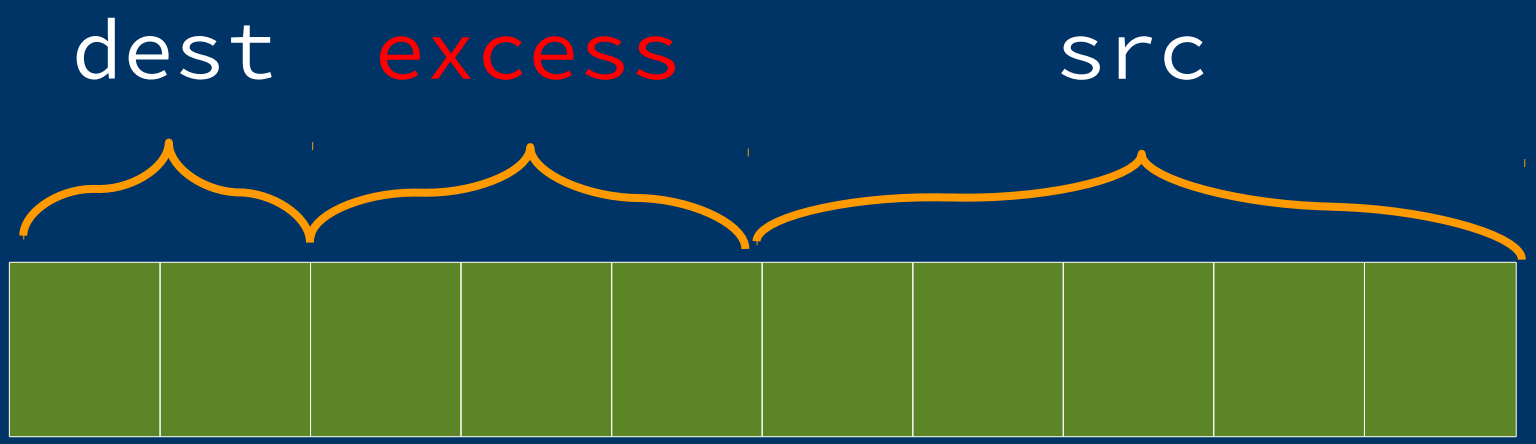
Strings can be defined as primitive types (Perl, Python), as an array of characters (C, C++), or as classes (Java, C++); string operations are usually implemented in some form.

EXAMPLE: C

Strings are `char` arrays terminated by a `'\0'` aka null character. String operations are provided in the `string.h` library.

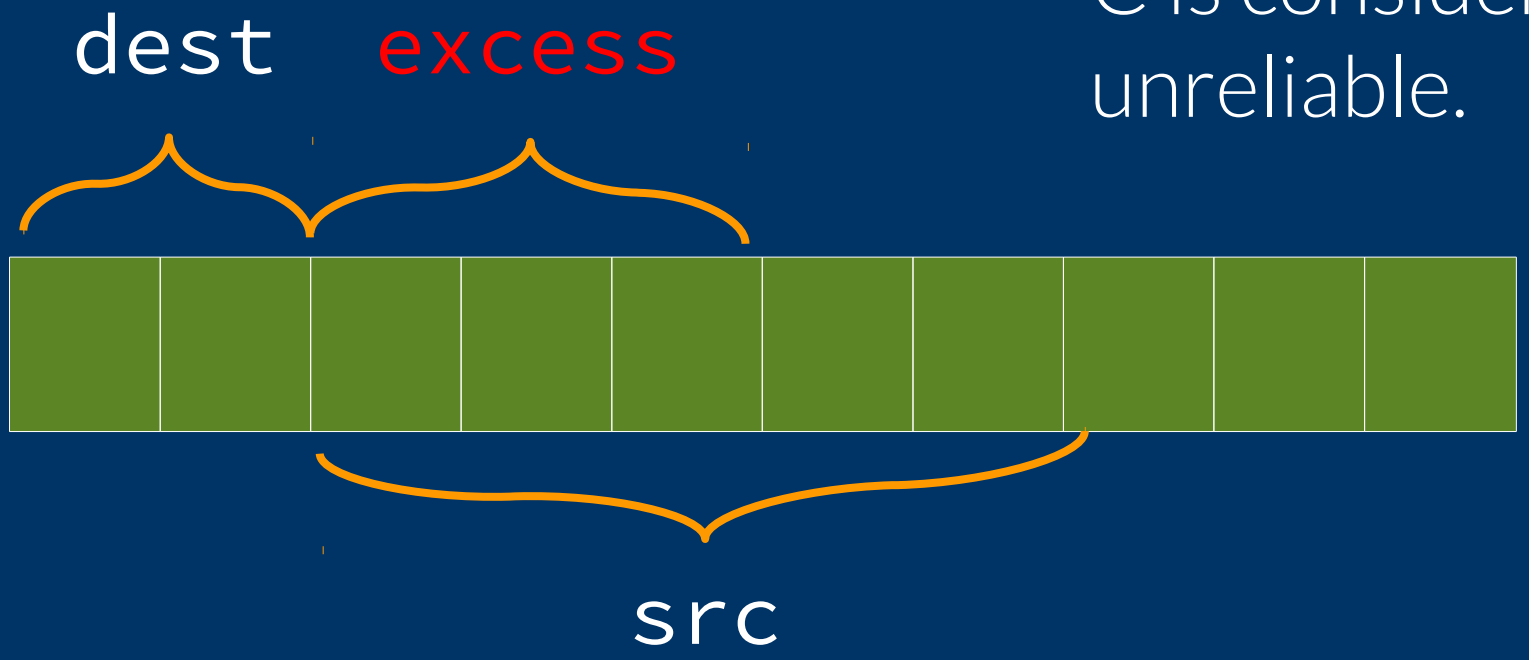
As seen in an earlier example, `string.h` operations do not guard against overflowing the destination string.

```
strcpy(dest, src);
```



But, what if...

```
strcpy(dest, src);
```



Which is one of the reasons why C is considered unreliable.

When given the option, use C++'s `string` class rather than C's `char []`.

Another issue about string length is whether it is static or dynamic.

STATIC LENGTH STRINGS

String length is fixed when the string is created.

PLs with static length strings are C++ (class version), Java, Ruby, and the .NET languages.

LIMITED DYNAMIC LENGTH STRINGS

String length can vary up to a maximum, set in the string declaration.

PLs with limited dynamic length strings are C, C++ (**char []** version), Java, Ruby, and the .NET languages.

DYNAMIC LENGTH STRINGS

String length may vary with no maximum.

Dynamic length strings are used in Perl, Javascript, among others.

CONSIDERATION

Though they provide maximum flexibility, the overhead of (de)allocation may be too much.

Strings have a big impact on PL writability, and are preferably implemented as a primitive type.

EXAMPLE

```
char str1[10],
str2[10];
...
for(int i=0;
i<10; i++)
  str2[i] =
    str1[i];
```

vs.

```
String str1,
str2;
...
str2 = str1;
```

C handles this by providing a string library (`string.h`).

Among string operations, simple pattern matching and concatenation are the most important.

Strings are usually
implemented in software
(not hardware-supported).

Aside from the descriptors described earlier, strings descriptors require:

1.

String length in characters

```
Ex. char str1[30] = “Hello world!”;
```

What is the length of str1?

In the case of limited dynamic length strings, an additional field for maximum length may be included.

2.

Address of the first character

Of course, there are exceptions...

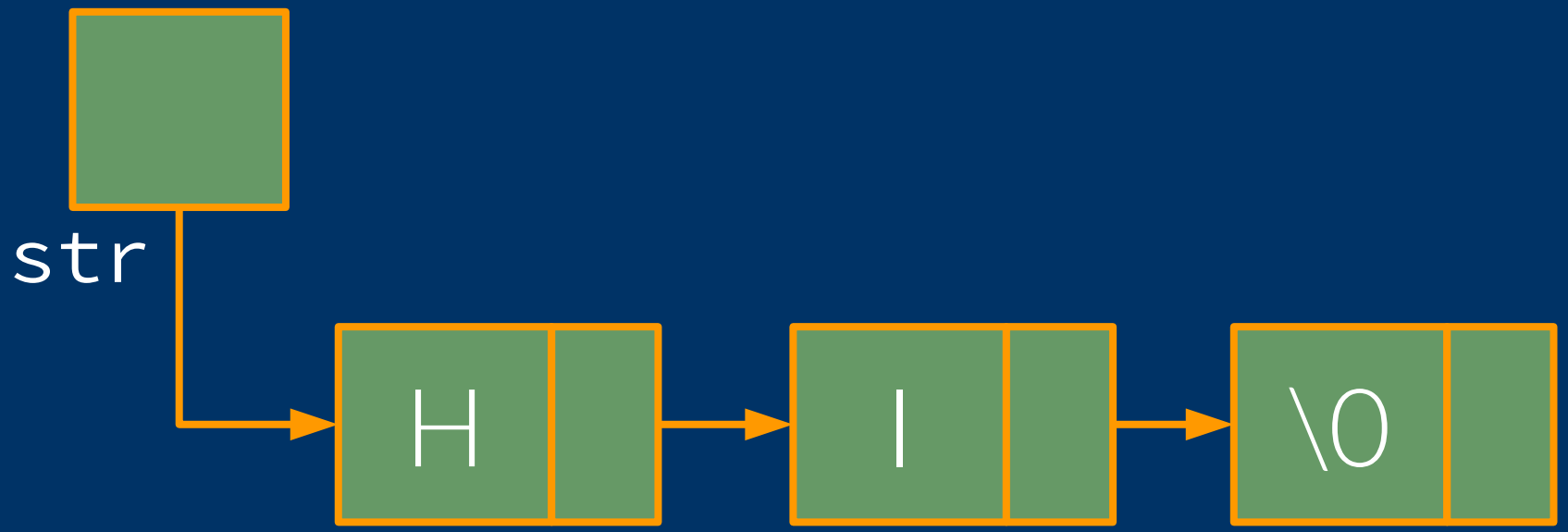
EXAMPLE: C

The end of the string depends on the location of the null character (`'\0'`).

Dynamic length strings may be implemented in three ways.

1.

Linked List



Linked list grows/shrinks as string length changes.

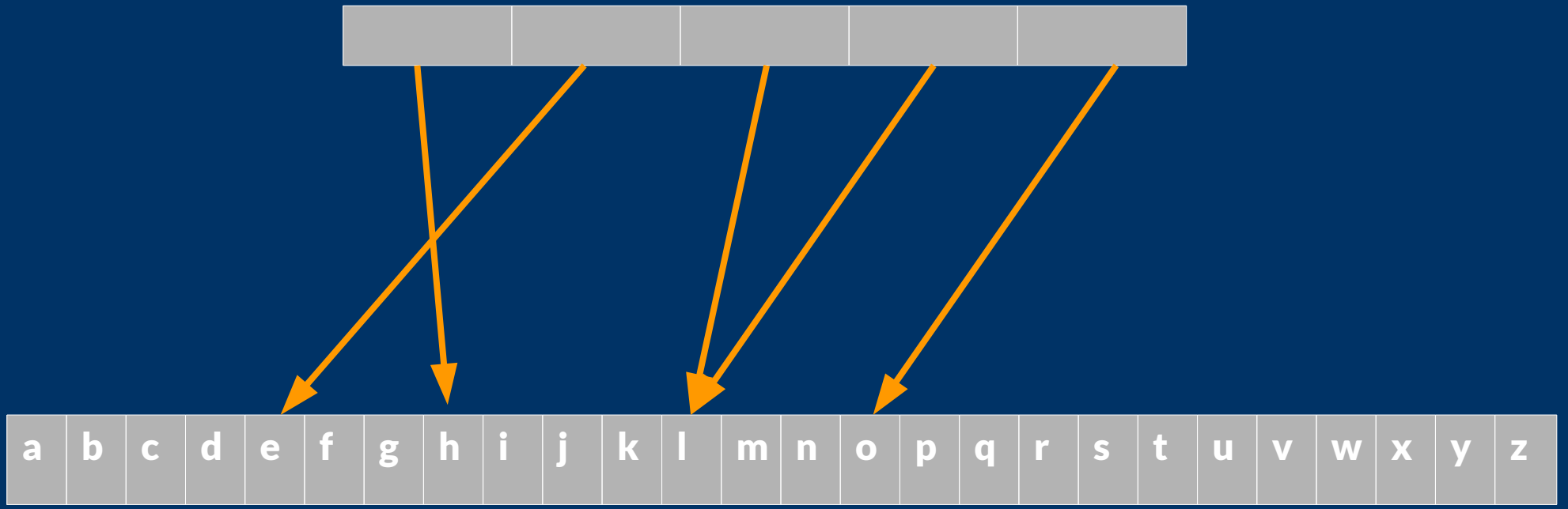
Though (de)allocation is simple, extra storage is required by the pointers used in linked lists.

Moreover, some string operations may be more complex when linked lists are used.

2.

Array of pointers to characters on the heap

“hello”



Though faster than the linked list implementation, it still uses extra memory.

3.

Store in adjacent memory cells

However, handling expanding length may be difficult.

To lengthen string, a new location of adjacent memory cells may need to be found; the old string is moved to the new location.

This is the string representation that is **most often used** (even by non-dynamic strings).

Fast string operations and less extra storage is used.

However, when string expansion happens, (de)allocation may slow things down.