EECS 22: Advanced C Programming

Lecture 18

Rainer Dömer

doemer@uci.edu

The Henry Samueli School of Engineering
Electrical Engineering and Computer Science
University of California, Irvine

Lecture 18: Overview

- Course Administration
  - Reminder: Final course evaluation
- Types
  - Type Conversion
  - Types in Expressions
  - Type Qualifiers
- Functions
  - Passing Data To/From Functions
  - Variable Argument Lists
- String Operations
- Standard Library
  - Functions defined in **`stdlib.h`**, **`string.h`**, **`math.h`**

Course Administration

- Final Course Evaluation
  - Open until end of 10th week (Sunday night)
  - Nov. 20, 2012, through Dec. 9, 2012, 11:45pm
  - Online via EEE Evaluation application
- Mandatory Evaluation of Course and Instructor
  - Voluntary
  - Anonymous
  - Very valuable
- Your feedback is appreciated!

Type Conversion

- Explicit Type Conversion
  - types can be explicitly converted to other types, by use of the type cast operator:
    (*type*) *expression*
  - the target type is named explicitly in parentheses before the source expression
  - Examples:
    - **`Float = (float) LongInt`**
      - converts the **`long int`** value into a **`float`** value
    - **`Integer = (int) Double`**
      - converts the **`double`** value into an **`int`** value
      - any fractional part is truncated!
    - **`Char = (char) LongLongInt`**
      - converts the **`long long int`** value into a **`char`** value
      - any out-of-range values are silently cut off!

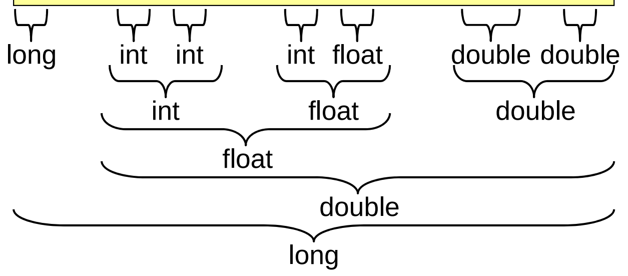
Type Conversion

- Implicit Type Conversion
  - Type promotion
    - integral promotion
      - **`unsigned`** or **`signed char`** is promoted to **`unsigned`** or **`signed int`** before any operation
      - **`unsigned`** or **`signed short`** is promoted to **`unsigned`** or **`signed int`** before any operation
    - binary arithmetic operators are defined only for same types
      - the smaller type is converted to the larger type (before operation)
      - Examples:
        - » **`ShortInt * LongInt`** results in a **`long int`** type
        - » **`LongDouble * Float`** results in a **`long double`** type
  - Type coercion
    - most types are automatically converted to expected types
    - Example: **`Double = Float`**, or **`Char = LongInt`**

Types in Expressions

- Expressions are composed of constants, variables and operators, each of which has an associated type
- Example:

```
short int s;
int       i;
long int  l;
float     f;
double    d;

l  =   2 * s   +   i * f   -   0.5   * d;
```

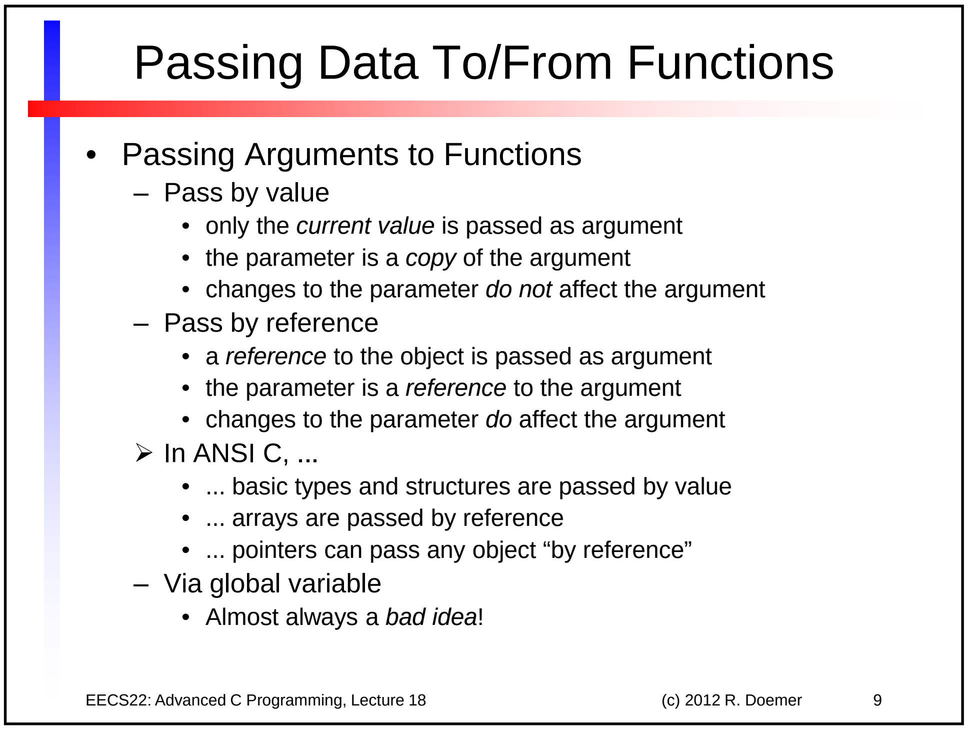


Type Qualifiers

- Types may be further qualified
  - Type qualifier **`const`**
    - The value of a **`const`** object cannot be changed
    - Initialization is OK, assignment is not
    - Example:
      - **`const double pi = 3.1415926536;`**
    - ➢ Object may be placed in read-only memory (ROM)
  - Type qualifier **`volatile`**
    - The value of a **`volatile`** object must not be used for compiler optimizations
    - Machine registers for memory-mapped I/O are volatile
    - Example:
      - **`volatile char *StatusReg = 0x40000000;`**
      - **`while(*StatusReg == 0x00) ;`**
    - ➢ Accesses to **`volatile`** objects must not be optimized away

Passing Data To/From Functions

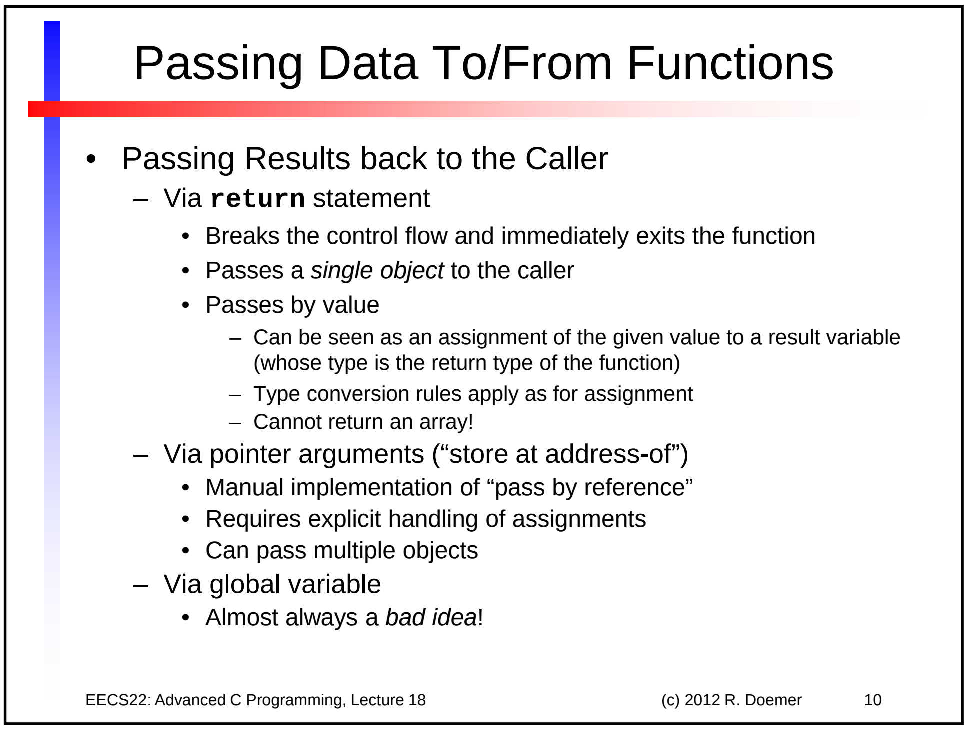
- Passing Arguments to Functions
  - Options:
    - Pass by value
    - Pass by reference
    - Via global variable
- Returning Results from Functions
  - Options:
    - Via return statement
    - Via pointer arguments (“store at address-of”)
    - Via global variable
- Considerations
  - Type of data (affects pass by value/reference)
  - Amount of data (affects performance)
  - Packaging in structures (**`struct`**)



Passing Data To/From Functions

- Passing Arguments to Functions
  - Pass by value
    - only the *current value* is passed as argument
    - the parameter is a *copy* of the argument
    - changes to the parameter *do not* affect the argument
  - Pass by reference
    - a *reference* to the object is passed as argument
    - the parameter is a *reference* to the argument
    - changes to the parameter *do* affect the argument
  - ➢ In ANSI C, ...
    - ... basic types and structures are passed by value
    - ... arrays are passed by reference
    - ... pointers can pass any object “by reference”
  - Via global variable
    - Almost always a *bad idea*!

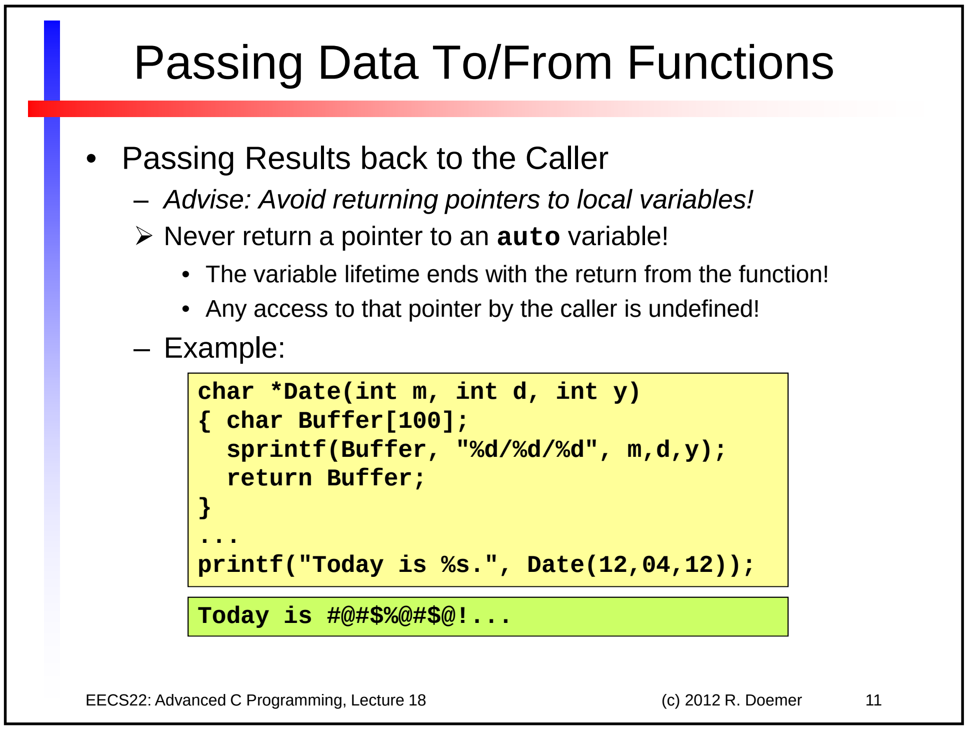
EECS22: Advanced C Programming, Lecture 18 (c) 2012 R. Doemer 9



Passing Data To/From Functions

- Passing Results back to the Caller
  - Via **`return`** statement
    - Breaks the control flow and immediately exits the function
    - Passes a *single object* to the caller
    - Passes by value
      - Can be seen as an assignment of the given value to a result variable (whose type is the return type of the function)
      - Type conversion rules apply as for assignment
      - Cannot return an array!
  - Via pointer arguments (“store at address-of”)
    - Manual implementation of “pass by reference”
    - Requires explicit handling of assignments
    - Can pass multiple objects
  - Via global variable
    - Almost always a *bad idea*!

EECS22: Advanced C Programming, Lecture 18 (c) 2012 R. Doemer 10



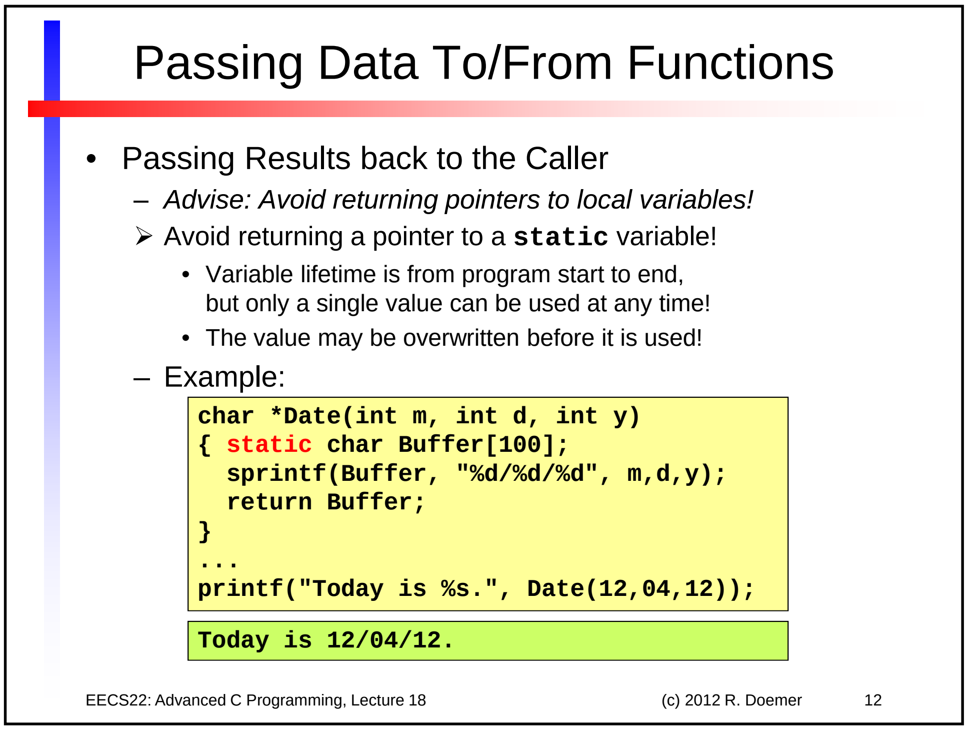
Passing Data To/From Functions

- Passing Results back to the Caller
  - *Advise: Avoid returning pointers to local variables!*
  - ➢ Never return a pointer to an **`auto`** variable!
    - The variable lifetime ends with the return from the function!
    - Any access to that pointer by the caller is undefined!
  - Example:

```
char *Date(int m, int d, int y)
{ char Buffer[100];
  sprintf(Buffer, "%d/%d/%d", m,d,y);
  return Buffer;
}
...
printf("Today is %s.", Date(12,04,12));
```

```
Today is #@#$%@#$@!...
```

EECS22: Advanced C Programming, Lecture 18 (c) 2012 R. Doemer 11



Passing Data To/From Functions

- Passing Results back to the Caller
  - *Advise: Avoid returning pointers to local variables!*
  - ➢ Avoid returning a pointer to a **`static`** variable!
    - Variable lifetime is from program start to end,
      but only a single value can be used at any time!
    - The value may be overwritten before it is used!
  - Example:

```
char *Date(int m, int d, int y)
{ static char Buffer[100];
  sprintf(Buffer, "%d/%d/%d", m,d,y);
  return Buffer;
}
...
printf("Today is %s.", Date(12,04,12));
```

```
Today is 12/04/12.
```

EECS22: Advanced C Programming, Lecture 18 (c) 2012 R. Doemer 12

Passing Data To/From Functions

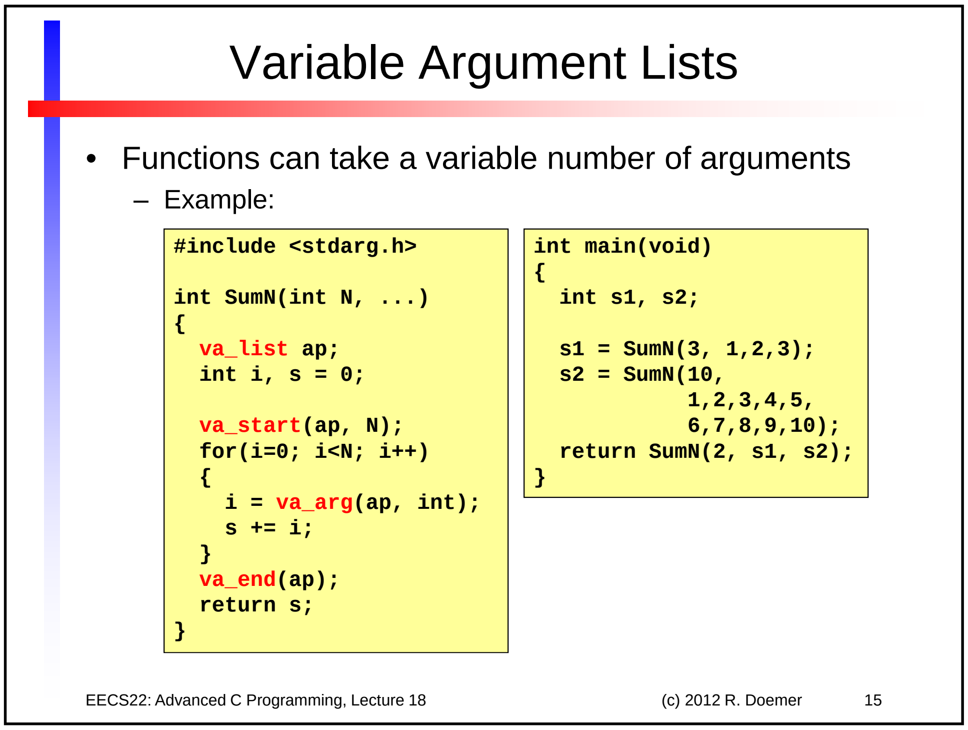
- Passing Results back to the Caller
  - *Advise: Avoid returning pointers to local variables!*
  - ➢ Avoid returning a pointer to a **`static`** variable!
    - Variable lifetime is from program start to end, but only a single value can be used at any time!
    - The value may be overwritten before it is used!
  - Example:

```
char *Date(int m, int d, int y)
{ static char Buffer[100];
  sprintf(Buffer, "%d/%d/%d", m,d,y);
  return Buffer;
}
...
printf("Today is %s, tomorrow is %s!",
       Date(12,04,12), Date(12,05,12));
```

```
Today is 12/05/12, tomorrow is 12/05/12!
```

Variable Argument Lists

- Functions can take a variable number of arguments
  - Example: **`int printf(char *fmt, ...);`**
  - Note: The declaration **`...`**
    - indicates a variable number of arguments are following
    - is a valid token of the C language
    - can be used only at the end of an argument list
  - Header file **`stdarg.h`** provides
    - Type **`va_list`**
      - Type of a pointer to an argument (e.g. **ap**)
    - Macro **`va_start(va_list ap, `*`last_arg`*`)`**
      - Initializes **ap** to point to the first variable argument after ***`last_arg`***
    - Macro **`va_arg(va_list ap, `*`type`*`)`**
      - Returns the value (of type ***`type`***) of the next variable argument
    - Macro **`va_end(va_list ap)`**
      - Must be called once after all arguments are processed but before the function exits



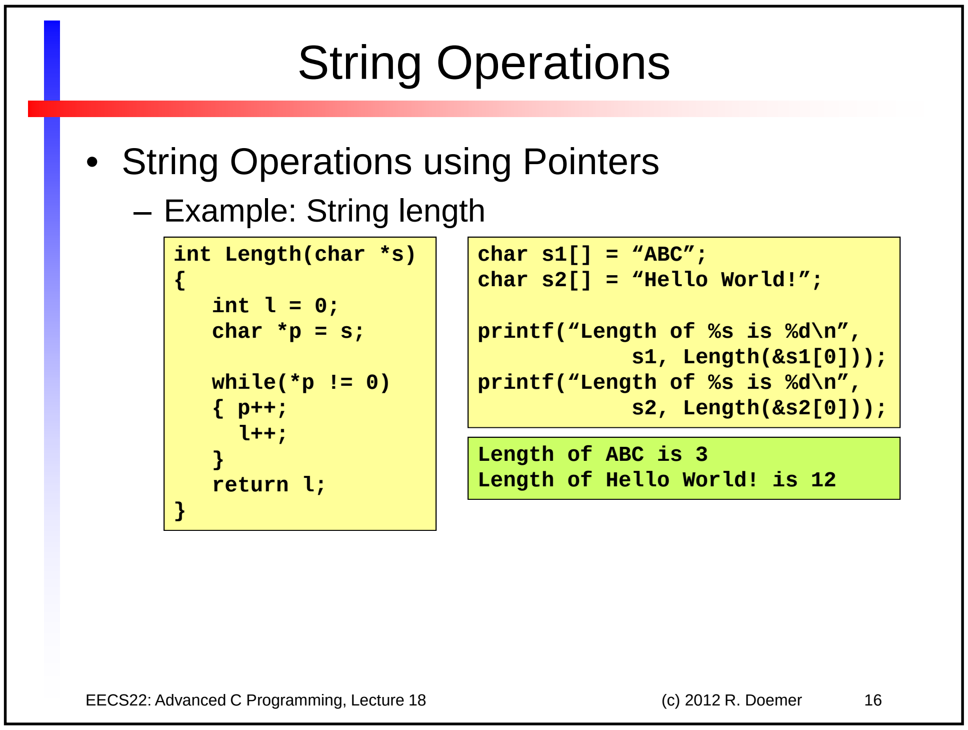
Variable Argument Lists

- Functions can take a variable number of arguments
  - Example:

```
#include <stdarg.h>

int SumN(int N, ...)
{
  va_list ap;
  int i, s = 0;

  va_start(ap, N);
  for(i=0; i<N; i++)
  {
    i = va_arg(ap, int);
    s += i;
  }
  va_end(ap);
  return s;
}
```

```
int main(void)
{
  int s1, s2;

  s1 = SumN(3, 1,2,3);
  s2 = SumN(10,
            1,2,3,4,5,
            6,7,8,9,10);
  return SumN(2, s1, s2);
}
```

EECS22: Advanced C Programming, Lecture 18 (c) 2012 R. Doemer 15



String Operations

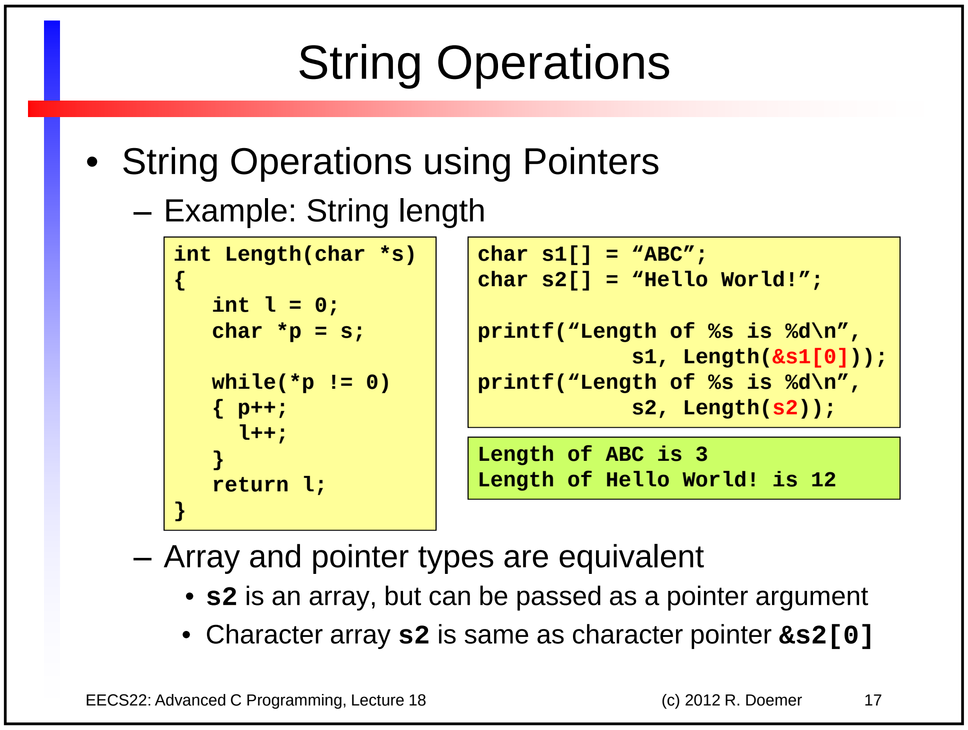
- String Operations using Pointers
  - Example: String length

```
int Length(char *s)
{
   int l = 0;
   char *p = s;

   while(*p != 0)
   { p++;
     l++;
   }
   return l;
}
```

```
char s1[] = “ABC”;
char s2[] = “Hello World!”;

printf(“Length of %s is %d\n”,
            s1, Length(&s1[0]));
printf(“Length of %s is %d\n”,
            s2, Length(&s2[0]));
```

```
Length of ABC is 3
Length of Hello World! is 12
```

EECS22: Advanced C Programming, Lecture 18 (c) 2012 R. Doemer 16



String Operations

- String Operations using Pointers
  - Example: String length

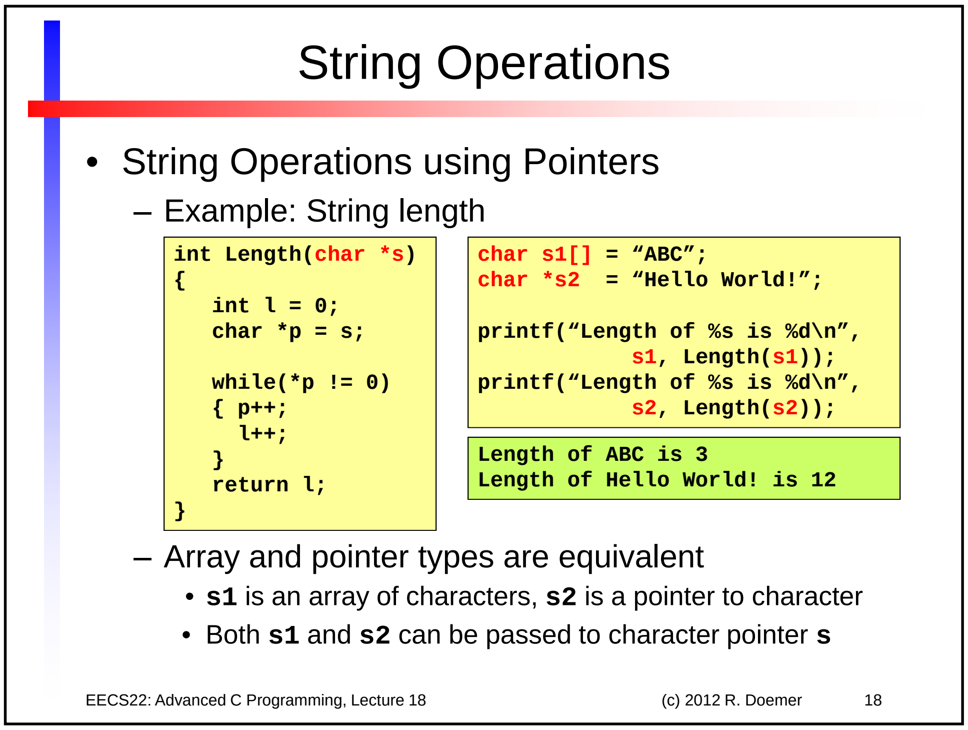
```
int Length(char *s)
{
   int l = 0;
   char *p = s;

   while(*p != 0)
   { p++;
     l++;
   }
   return l;
}
```

```
char s1[] = “ABC”;
char s2[] = “Hello World!”;

printf(“Length of %s is %d\n”,
            s1, Length(&s1[0]));
printf(“Length of %s is %d\n”,
            s2, Length(s2));
```

```
Length of ABC is 3
Length of Hello World! is 12
```

  - Array and pointer types are equivalent
    - **s2** is an array, but can be passed as a pointer argument
    - Character array **s2** is same as character pointer **&s2[0]**

EECS22: Advanced C Programming, Lecture 18 (c) 2012 R. Doemer 17



String Operations

- String Operations using Pointers
  - Example: String length

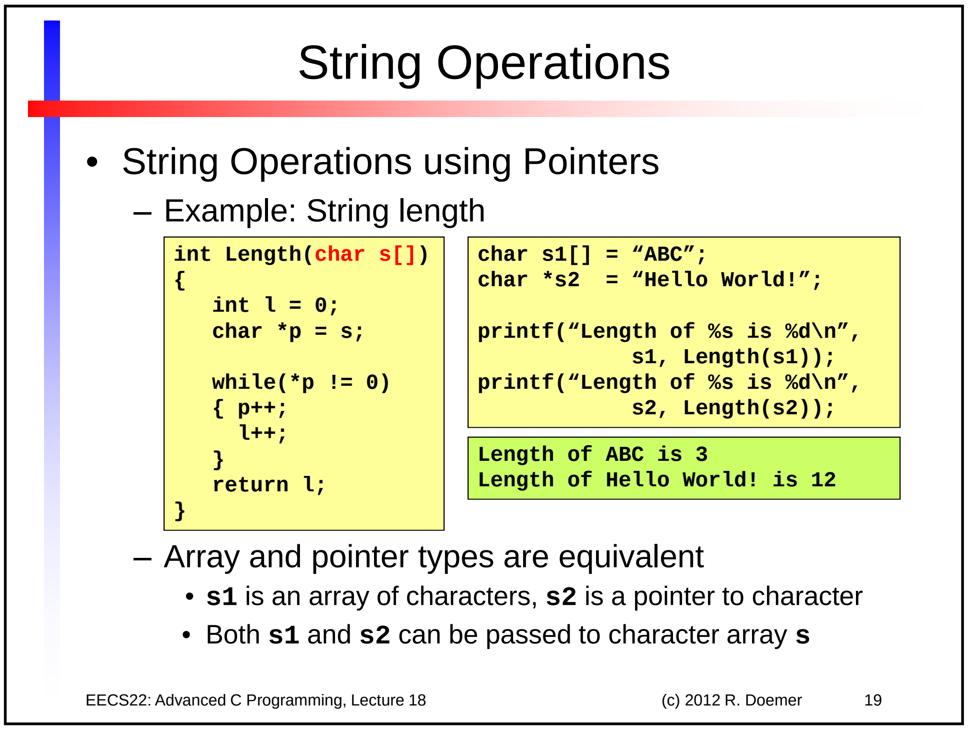
```
int Length(char *s)
{
   int l = 0;
   char *p = s;

   while(*p != 0)
   { p++;
     l++;
   }
   return l;
}
```

```
char s1[] = “ABC”;
char *s2  = “Hello World!”;

printf(“Length of %s is %d\n”,
            s1, Length(s1));
printf(“Length of %s is %d\n”,
            s2, Length(s2));
```

```
Length of ABC is 3
Length of Hello World! is 12
```

  - Array and pointer types are equivalent
    - **s1** is an array of characters, **s2** is a pointer to character
    - Both **s1** and **s2** can be passed to character pointer **s**

EECS22: Advanced C Programming, Lecture 18 (c) 2012 R. Doemer 18



String Operations

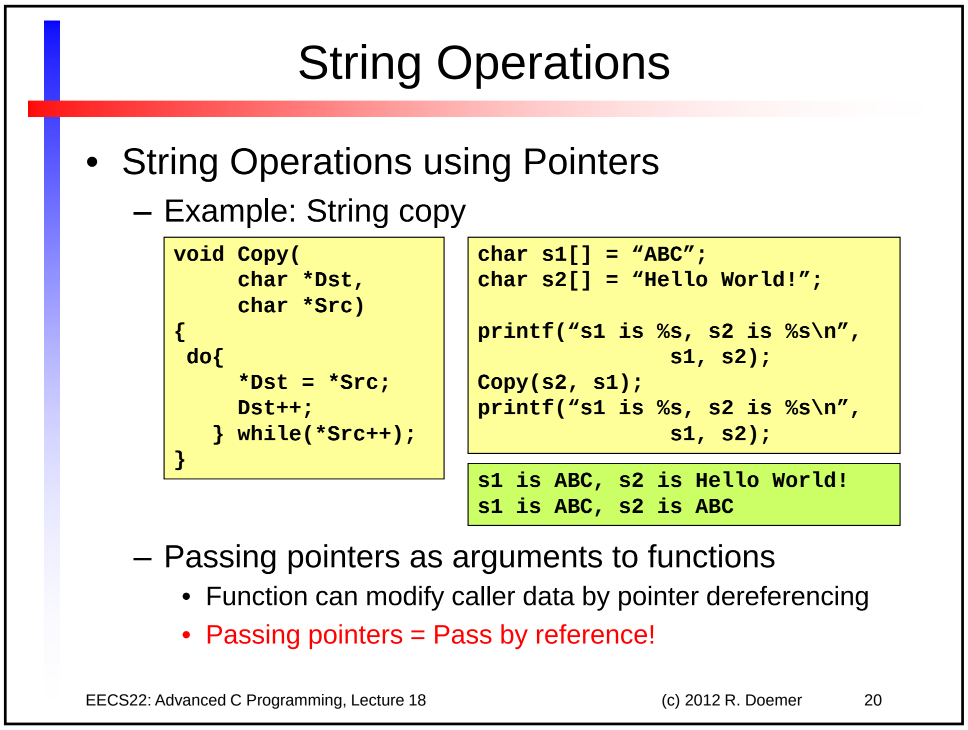
- String Operations using Pointers
  - Example: String length

```
int Length(char s[])
{
   int l = 0;
   char *p = s;

   while(*p != 0)
   { p++;
     l++;
   }
   return l;
}
```

```
char s1[] = “ABC”;
char *s2  = “Hello World!”;

printf(“Length of %s is %d\n”,
            s1, Length(s1));
printf(“Length of %s is %d\n”,
            s2, Length(s2));
```

```
Length of ABC is 3
Length of Hello World! is 12
```

  - Array and pointer types are equivalent
    - **s1** is an array of characters, **s2** is a pointer to character
    - Both **s1** and **s2** can be passed to character array **s**



String Operations

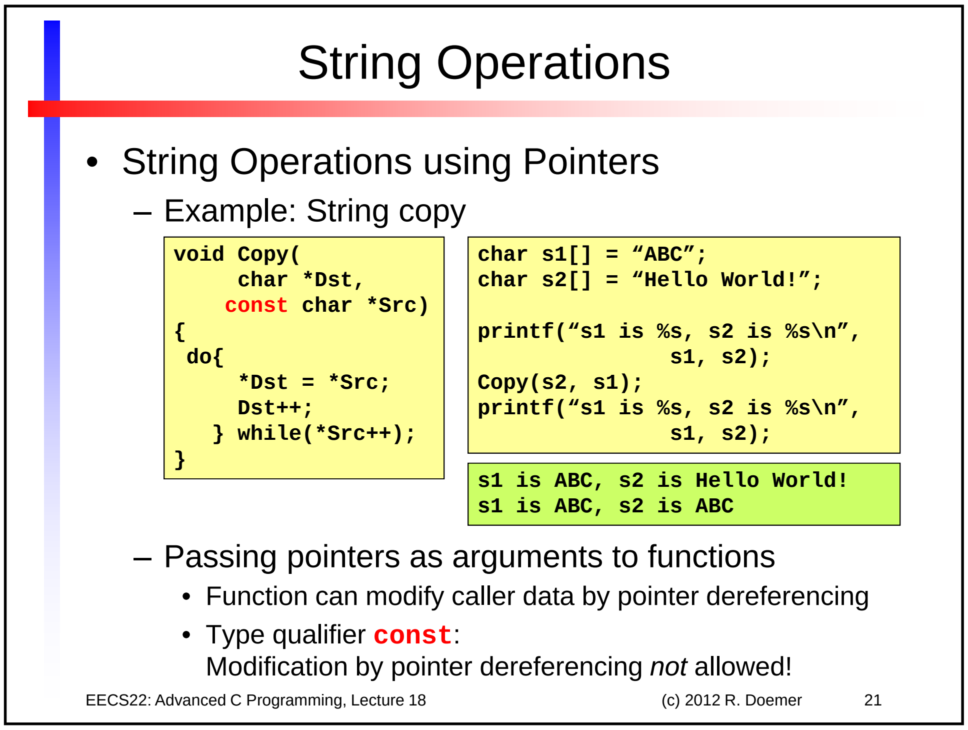
- String Operations using Pointers
  - Example: String copy

```
void Copy(
     char *Dst,
     char *Src)
{
 do{
     *Dst = *Src;
     Dst++;
   } while(*Src++);
}
```

```
char s1[] = “ABC”;
char s2[] = “Hello World!”;

printf(“s1 is %s, s2 is %s\n”,
               s1, s2);
Copy(s2, s1);
printf(“s1 is %s, s2 is %s\n”,
               s1, s2);
```

```
s1 is ABC, s2 is Hello World!
s1 is ABC, s2 is ABC
```

  - Passing pointers as arguments to functions
    - Function can modify caller data by pointer dereferencing
    - Passing pointers = Pass by reference!



String Operations

- String Operations using Pointers
  - Example: String copy

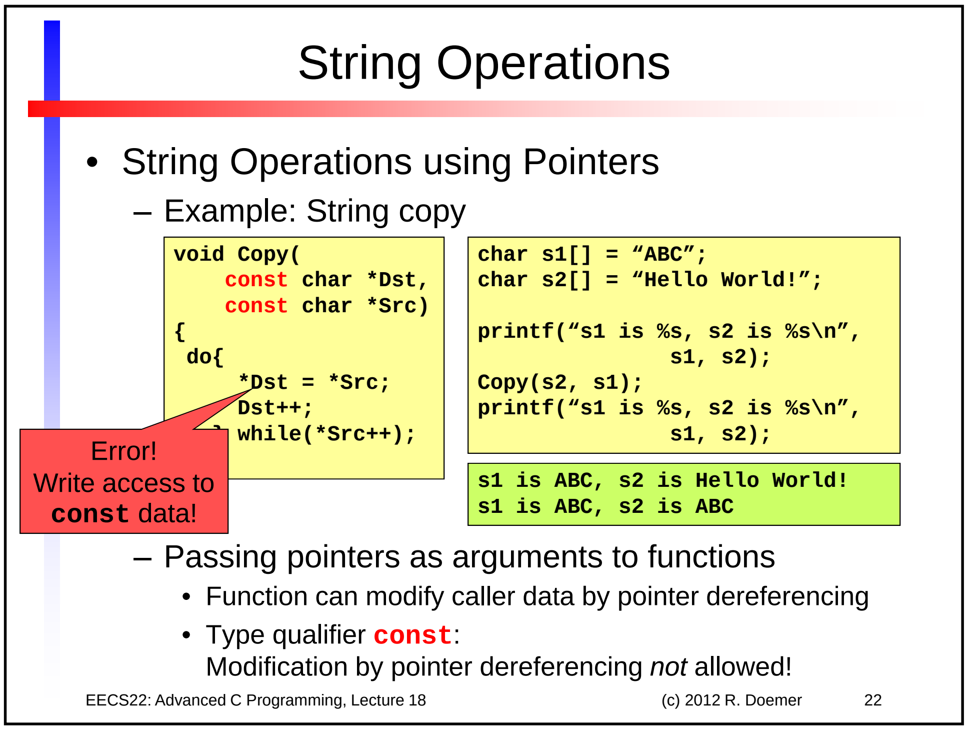
```
void Copy(
     char *Dst,
    const char *Src)
{
 do{
     *Dst = *Src;
     Dst++;
   } while(*Src++);
}
```

```
char s1[] = “ABC”;
char s2[] = “Hello World!”;

printf(“s1 is %s, s2 is %s\n”,
               s1, s2);
Copy(s2, s1);
printf(“s1 is %s, s2 is %s\n”,
               s1, s2);
```

```
s1 is ABC, s2 is Hello World!
s1 is ABC, s2 is ABC
```

  - Passing pointers as arguments to functions
    - Function can modify caller data by pointer dereferencing
    - Type qualifier **`const`**:
      Modification by pointer dereferencing *not* allowed!

EECS22: Advanced C Programming, Lecture 18 (c) 2012 R. Doemer 21



String Operations

- String Operations using Pointers
  - Example: String copy

```
void Copy(
    const char *Dst,
    const char *Src)
{
 do{
     *Dst = *Src;
     Dst++;
   } while(*Src++);
}
```

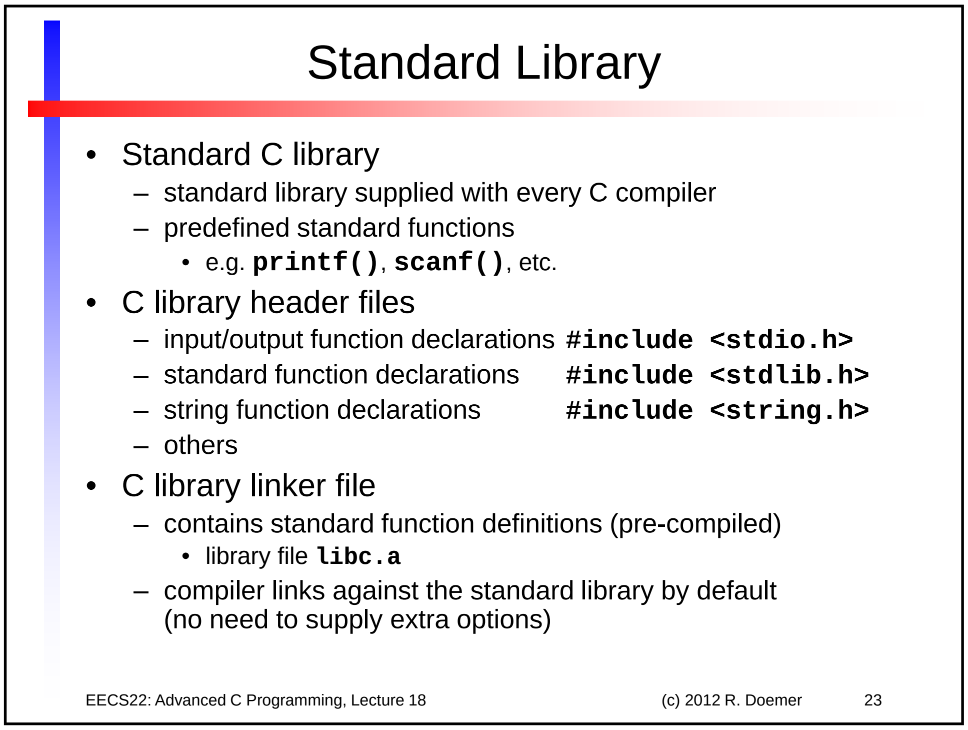
Error!
Write access to
`const` data!

```
char s1[] = “ABC”;
char s2[] = “Hello World!”;

printf(“s1 is %s, s2 is %s\n”,
               s1, s2);
Copy(s2, s1);
printf(“s1 is %s, s2 is %s\n”,
               s1, s2);
```

```
s1 is ABC, s2 is Hello World!
s1 is ABC, s2 is ABC
```

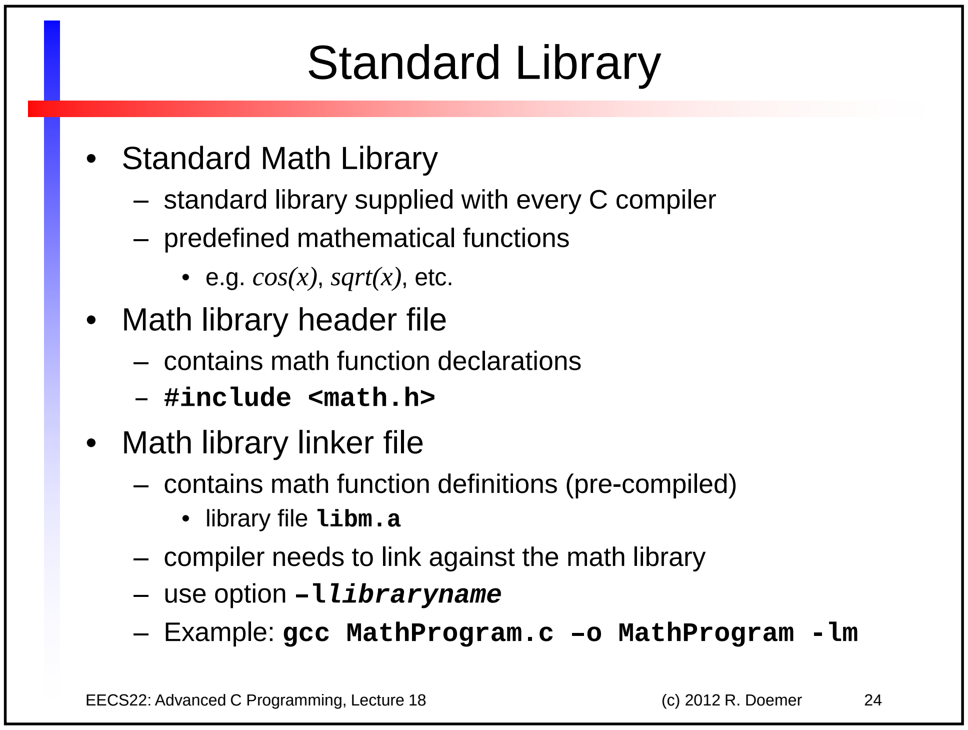
  - Passing pointers as arguments to functions
    - Function can modify caller data by pointer dereferencing
    - Type qualifier **`const`**:
      Modification by pointer dereferencing *not* allowed!

EECS22: Advanced C Programming, Lecture 18 (c) 2012 R. Doemer 22



Standard Library

- Standard C library
  - standard library supplied with every C compiler
  - predefined standard functions
    - e.g. **`printf()`**, **`scanf()`**, etc.
- C library header files
  - input/output function declarations **`#include <stdio.h>`**
  - standard function declarations **`#include <stdlib.h>`**
  - string function declarations **`#include <string.h>`**
  - others
- C library linker file
  - contains standard function definitions (pre-compiled)
    - library file **`libc.a`**
  - compiler links against the standard library by default (no need to supply extra options)



Standard Library

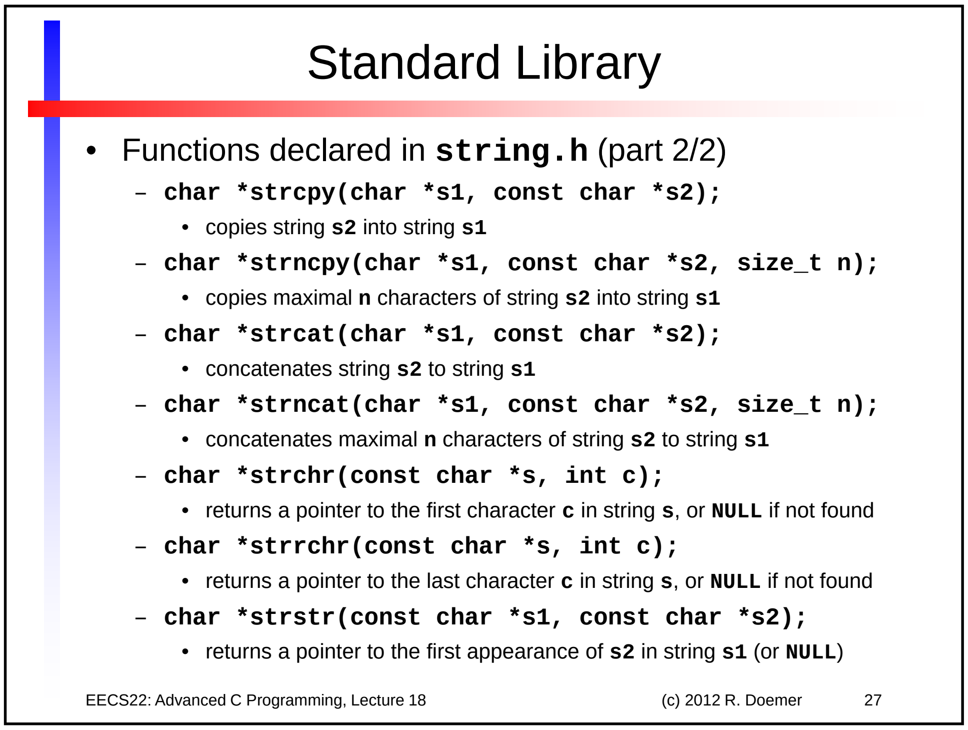
- Standard Math Library
  - standard library supplied with every C compiler
  - predefined mathematical functions
    - e.g. *cos(x)*, *sqrt(x)*, etc.
- Math library header file
  - contains math function declarations
  - **`#include <math.h>`**
- Math library linker file
  - contains math function definitions (pre-compiled)
    - library file **`libm.a`**
  - compiler needs to link against the math library
  - use option **`–l`*`libraryname`***
  - Example: **`gcc MathProgram.c –o MathProgram -lm`**

Standard Library

- Functions declared in **`stdlib.h`** (selected subset)
  - **`int abs(int x);`**
  - **`long int labs(long int x);`**
    - return the absolute value of a (long) integer **`x`**
  - **`int rand(void);`**
    - return a random value in the range **`0 – RAND_MAX`**
    - **`RAND_MAX`** is a constant integer (e.g. 32767)
  - **`void srand(unsigned int seed);`**
    - initialize the random number generator with value **`seed`**
  - **`void exit(int result);`**
    - exit the program with return value **`result`**
  - **`void abort(void);`**
    - abort the program (with an error result)

Standard Library

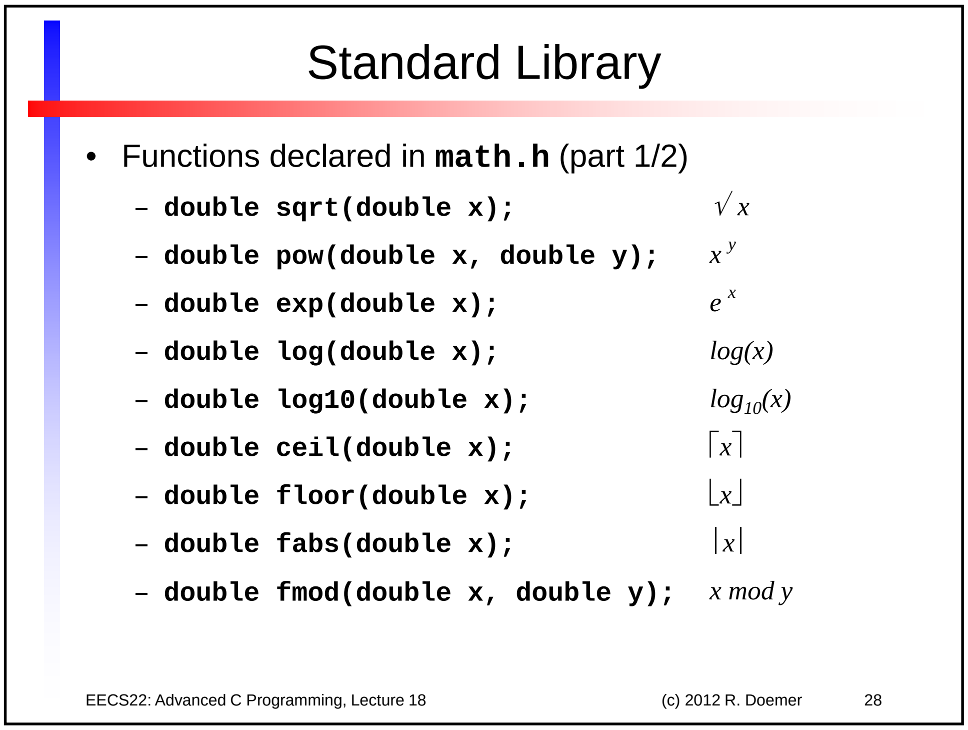
- Functions declared in **`string.h`** (part 1/2)
  - **`typedef unsigned int size_t;`**
    - type definition for length of strings
  - **`size_t strlen(const char *s);`**
    - returns the length of string **`s`**
  - **`int strcmp(const char *s1, const char *s2);`**
    - alphabetically compares string **`s1`** with string **`s2`**
    - returns -1 / 0 / 1 for less-than / equal-to / greater-than
  - **`int strncmp(const char *s1, const char *s2, size_t n);`**
    - same as previous, but compares maximal **`n`** characters
  - **`int strcasecmp(const char *s1, const char *s2);`**
  - **`int strncasecmp(const char *s1, const char *s2, size_t n);`**
    - same as string comparisons above, but case-insensitive



Standard Library

- Functions declared in **`string.h`** (part 2/2)
  - **`char *strcpy(char *s1, const char *s2);`**
    - copies string **`s2`** into string **`s1`**
  - **`char *strncpy(char *s1, const char *s2, size_t n);`**
    - copies maximal **`n`** characters of string **`s2`** into string **`s1`**
  - **`char *strcat(char *s1, const char *s2);`**
    - concatenates string **`s2`** to string **`s1`**
  - **`char *strncat(char *s1, const char *s2, size_t n);`**
    - concatenates maximal **`n`** characters of string **`s2`** to string **`s1`**
  - **`char *strchr(const char *s, int c);`**
    - returns a pointer to the first character **`c`** in string **`s`**, or **`NULL`** if not found
  - **`char *strrchr(const char *s, int c);`**
    - returns a pointer to the last character **`c`** in string **`s`**, or **`NULL`** if not found
  - **`char *strstr(const char *s1, const char *s2);`**
    - returns a pointer to the first appearance of **`s2`** in string **`s1`** (or **`NULL`**)

EECS22: Advanced C Programming, Lecture 18 (c) 2012 R. Doemer 27

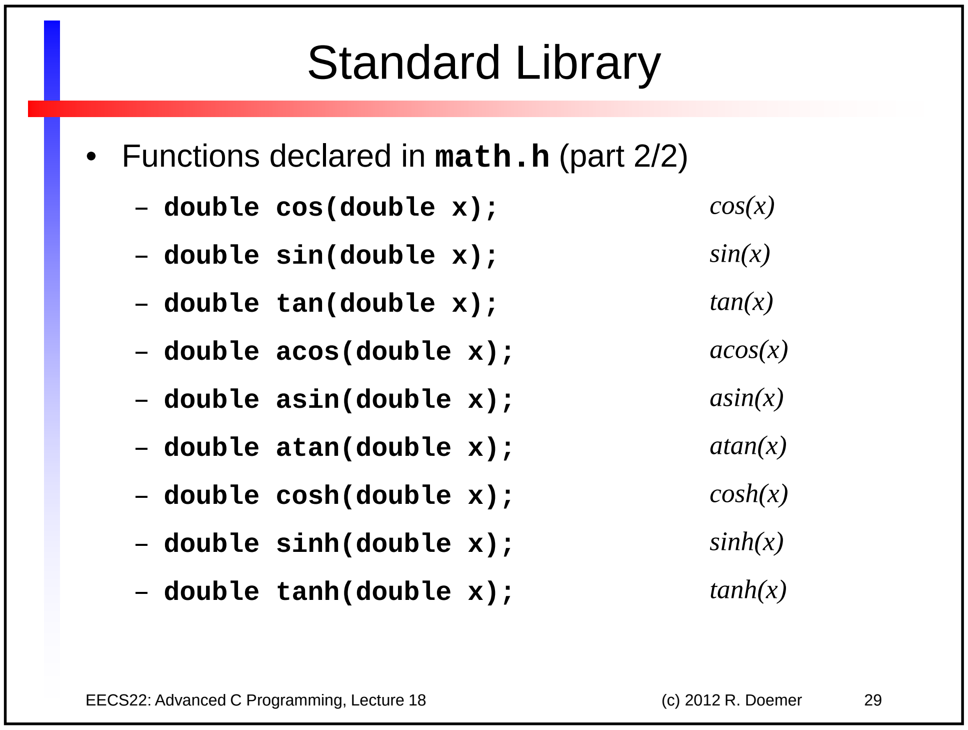


Standard Library

- Functions declared in **`math.h`** (part 1/2)

– **`double sqrt(double x);`**	$\sqrt{x}$
– **`double pow(double x, double y);`**	x^y
– **`double exp(double x);`**	e^x
– **`double log(double x);`**	$log(x)$
– **`double log10(double x);`**	$log_{10}(x)$
– **`double ceil(double x);`**	$\lceil x \rceil$
– **`double floor(double x);`**	$\lfloor x \rfloor$
– **`double fabs(double x);`**	$\lvert x \rvert$
– **`double fmod(double x, double y);`**	$x \bmod y$

EECS22: Advanced C Programming, Lecture 18 (c) 2012 R. Doemer 28



Standard Library

- Functions declared in **`math.h`** (part 2/2)

– **`double cos(double x);`**	*cos(x)*
– **`double sin(double x);`**	*sin(x)*
– **`double tan(double x);`**	*tan(x)*
– **`double acos(double x);`**	*acos(x)*
– **`double asin(double x);`**	*asin(x)*
– **`double atan(double x);`**	*atan(x)*
– **`double cosh(double x);`**	*cosh(x)*
– **`double sinh(double x);`**	*sinh(x)*
– **`double tanh(double x);`**	*tanh(x)*