C++ Programming: Functions

Domingos Begalli & Nadia Ahmed
Based on Work of Davender S. Malik

Saddleback College, Computer Science

CS1B

C++ Functions

we will cover

- predefined functions
- user-defined functions
- value-returning functions
- function prototype
- `void` functions
- formal and actual parameters
- value and reference parameters

C++ Functions

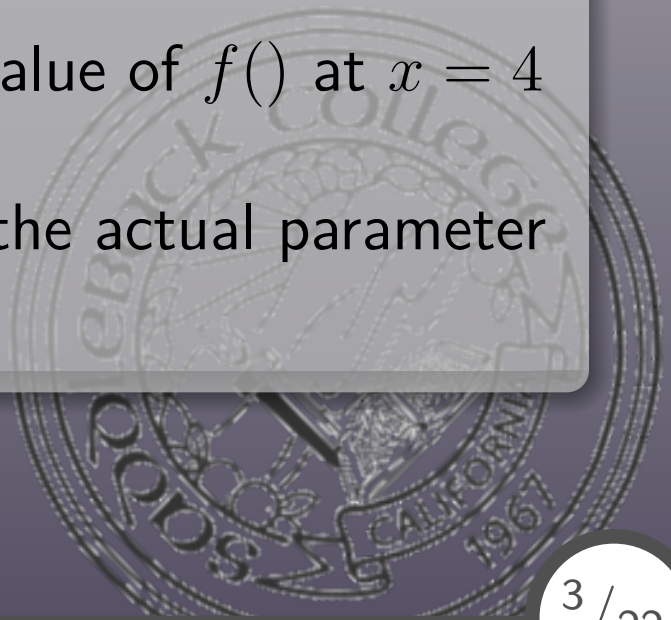
Basics

in math...

$$f(x) = 3\sqrt{x} - 1$$

$$f(4) = 3 \times \sqrt{4} - 1 = 5$$

- 4 is the argument of $f()$ and 5 the value of $f()$ at $x = 4$
- in c++ lingo, x is the formal and 4 the actual parameter



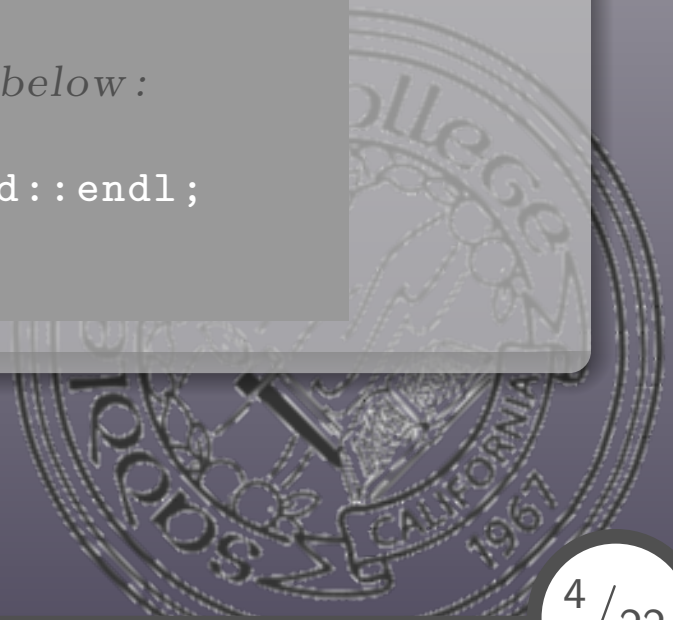
C++ Functions

Pre-Defined Functions

in a c++ predefined function - sqrt()

```
#include <iostream>
#include <cmath>

int main() {
  double x = 4;
  double y;
  // x is the actual parameter below:
  y = 3 * sqrt(x) - 1;
  std::cout << "y: " << y << std::endl;
  return 0;
}
```



C++ Functions

Function Libraries

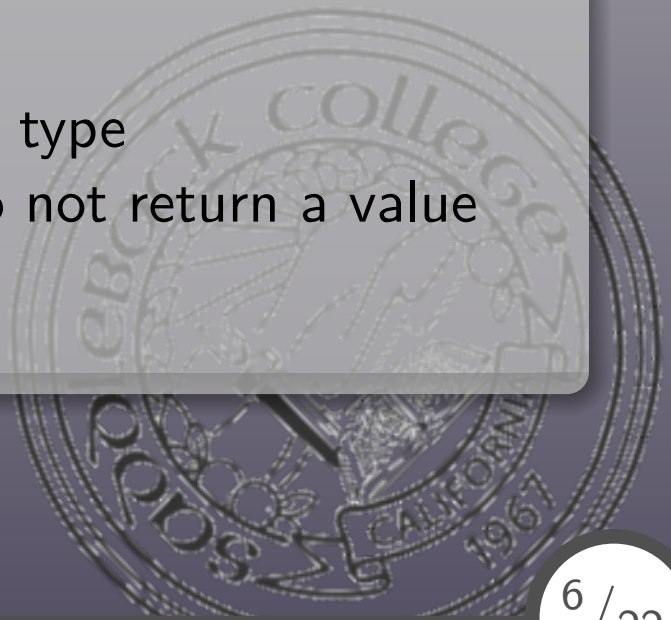
function	header file	description	parameter	result type
`abs(x)`	`<cmath>`	absolute value of `x`	`int,double`	`int,double`
`ceil(x)`	`<cmath>`	next whole number	`double`	`double`
`floor(x)`	`<cmath>`	previous whole number	`double`	`double`
`pow(x,y)`	`<cmath>`	returns x^y	`double`	`double`
`exp(x)`	`<cmath>`	returns e^x	`double`	`double`
`sqrt(x)`	`<cmath>`	returns $+\sqrt{x}$	`double`	`double`
`cos(x)`	`<cmath>`	cosine of `x`	`double`	`double`
`islower(x)`	`<cctype>`	returns `1` if `x` is lowercase	`int`	`int`
`isupper(x)`	`<cctype>`	returns `1` if `x` is uppercase	`int`	`int`
`tolower(x)`	`<cctype>`	returns lowercase value of `x`	`int`	`int`
`toupper(x)`	`<cctype>`	returns uppercase value of `x`	`int`	`int`
`time(x)`	`<ctime>`	current time	`time_t`	`time_t`

C++ Functions

User-Defined Functions

can return a value or not

- value returning functions
  - functions that have a `return` type
  - `return` a value of specific type
- `void` functions
  - functions that do not have a `return` type
  - can have a `return` statement but do not return a value



C++ Functions

User-Defined Functions

c++ user-defined function syntax

function definition:

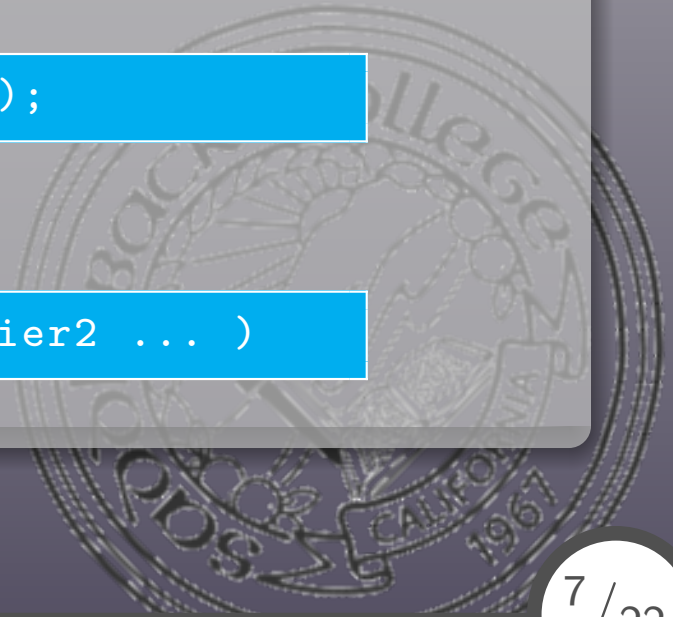
```
Type funcName(formal parameters) {
    statements
}
```

function call:

```
funcName(actual parameters);
```

parameters:

```
(Type1 identifier1, Type2 identifier2 ... )
```



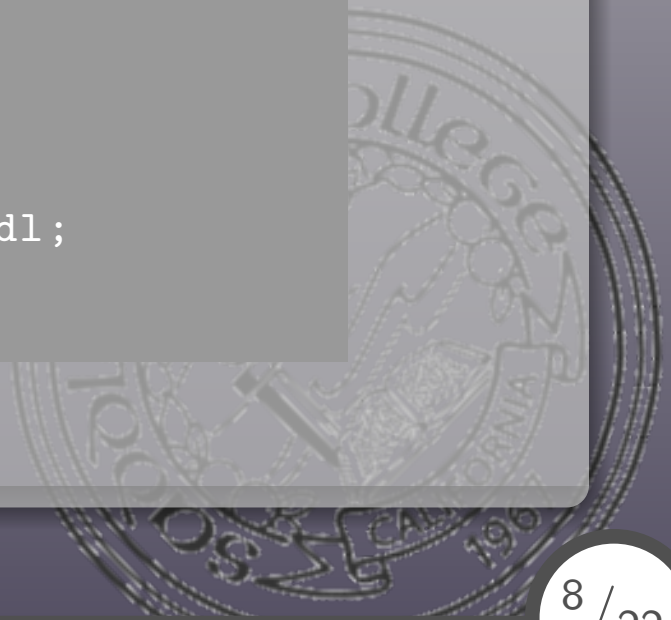
C++ Functions

User-Defined Functions

algebra.. and corresponding c++ user-defined function

$$f(x) = 3x^2 - 1; \quad f(4) = 3 \times 4^2 - 1 = 47$$

```
#include <iostream>
double calcsq(double x) {
  return 3 * x * x - 1;
}
int main() {
  double y, x = 4;
  y = calcsq(x);
  std::cout << "y: " << y << std::endl;
  return 0;
} // prints "y: 47"
```



C++ Functions

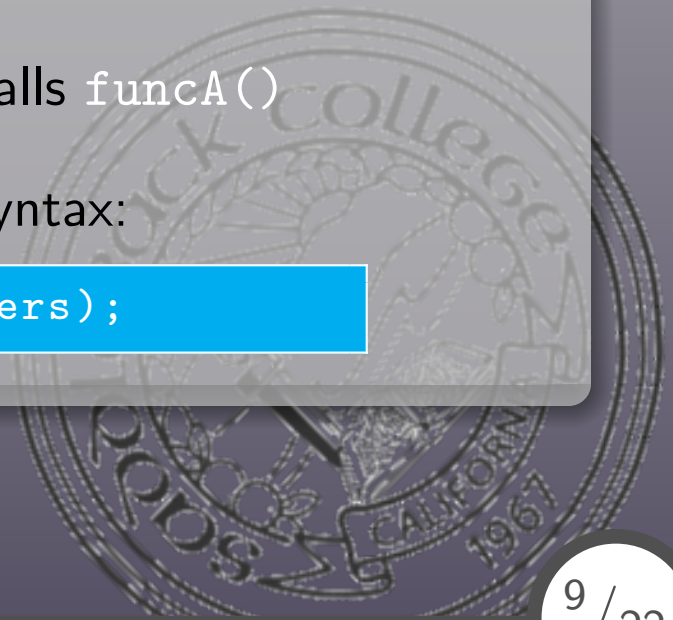
User-Defined Functions

function prototype/declaration

- the compiler issues an error if the function does not appear before `main()`
- a function declaration before `main()` keeps the compiler happy
- allows the function to be defined anywhere, including in an external file
- also resolves mutual calls:
  - when `funcA()` calls `funcB()` and `funcB()` calls `funcA()`

function prototype/declaration syntax:

```
Type funcName(formal parameters);
```



C++ Functions

User-Defined Functions

function prototype

```
#include <iostream>

double calcsq(double); // function prototype

int main() {
  double y, x = 4;
  y = calcsq(x);
  std::cout << "y: " << y << std::endl;
  return 0;
} // prints y: 47

// function definition after main():
double calcsq(double x) {
  return 3 * x * x - 1;
}
```

C++ Functions

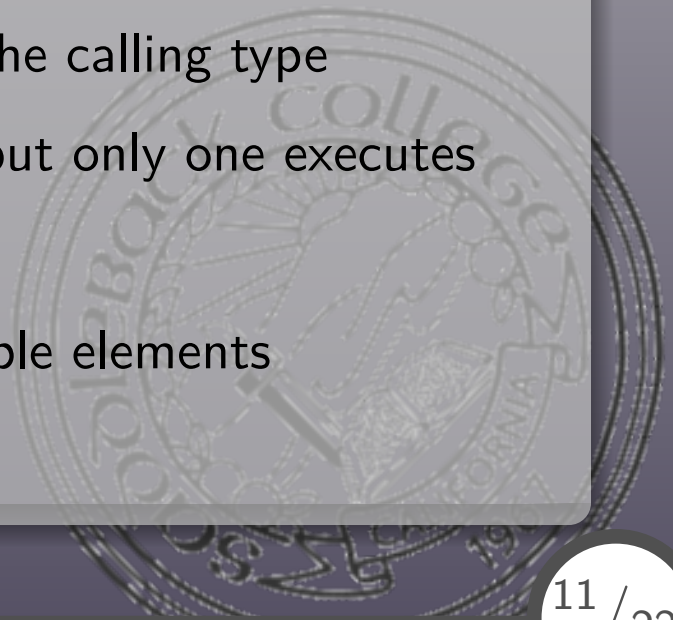
User-Defined Functions

the `return` statement

```
return expr;
```

where `expr` is either a variable, constant value, or expression

- the data type computed by `expr` has to match the calling type
- there might be multiple `return`s in a function, but only one executes
- a function can `return` only one value
- such `return`ed value can, in turn, address multiple elements



void functions

- functions that do not `return` a value
- have no type associated with them
  - c++11 defines the type `void` however
- have two types of parameters:
  - **value**: formal parameter receives a copy of actual parameter

```
void funcName(type1 var1, type2 var2, ... );
```

  - **reference**: formal parameter receives the address instead:

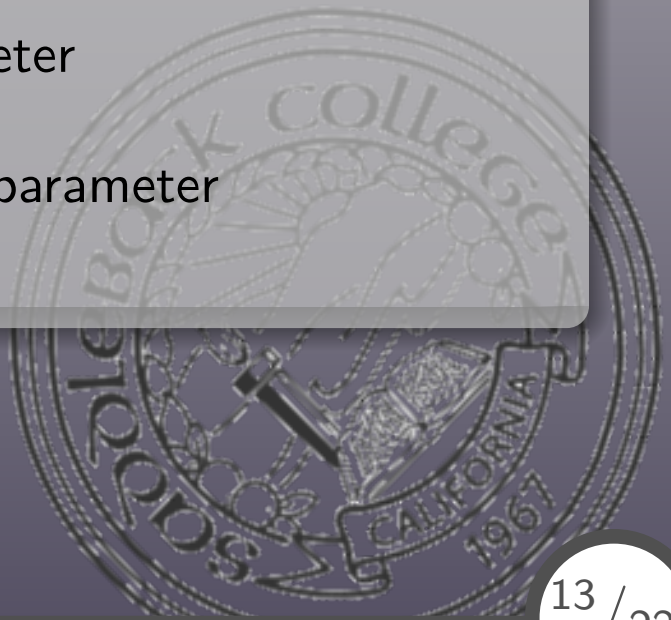
```
void funcName(type1& var1, type2& var2, ... );
```

C++ Functions

Parameter Types

value parameters

- formal parameters do not affect actual parameters
- actual parameters have no knowledge of formal parameters
- the function receives a copy of the actual parameter
- the function manipulates the copy of the actual parameter

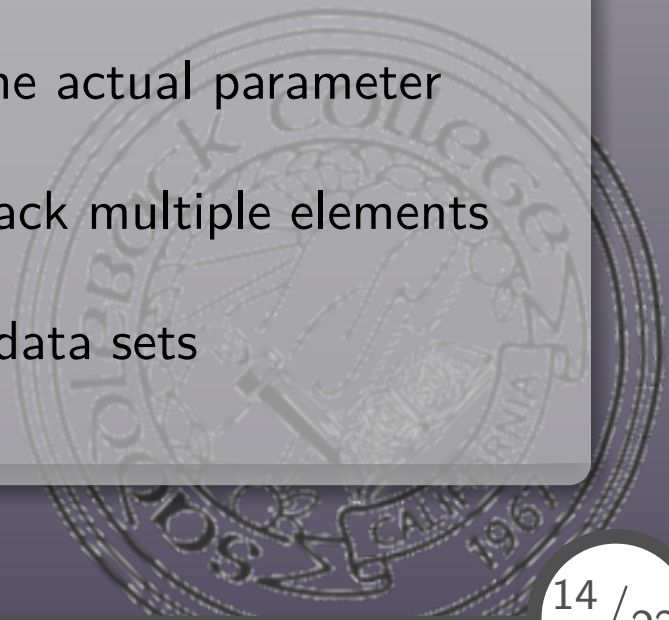


C++ Functions

Parameter Types

reference parameters

- formal parameters DO affect actual parameters
- formal parameters receive the address of actual parameters
- in this case, formal parameters are also called reference parameters
- the function manipulates the memory space of the actual parameter
- using reference parameters, functions can pass back multiple elements
- very usefull when calculations involve very large data sets



C++ Functions

Scope of Identifiers

local vs global identifiers

- **local**: identifiers declared within functions or code blocks
- **global**: identifiers declared ouside of function definitions

for example:

```
...
for (int i = 0; i < 10; ++i) {
  cout << "i: " << i << endl;
}
...
```

the variable `i` is not visible/defined outside the `for()` loop

C++ Functions

Types of Variables

`static` vs automatic variable

- **local** variables are automatic variables
- memory for automatic variables is allocated at block entry
- memory for automatic variables is deallocated at block exit
- **global** variables are `static` variables
- memory for `static` and **global** variables remains allocated during program execution
- the scope of `static` variables is limited to the code block
- therefore `static` variables are NOT **global**

declaring `static` variables:

```
static typeName varName;
```

C++ Functions

Types of Variables

global vs local vs `static` vs automatic

- **local** & **global** are related to visibility scope whereas
- `static` & automatic are related to memory allocation/deallocation:

```
int main() {
  cout << "notice x changes:\n";
  for(int ct=1; ct<=9; ct++)
    test();
}
void test() {
  static int x = 0;
  int y = 0;
  x = x + 1;
  y = y + 1;
  cout << "x: " << x ;
  cout << ", y: " << y << endl;
}
```

output

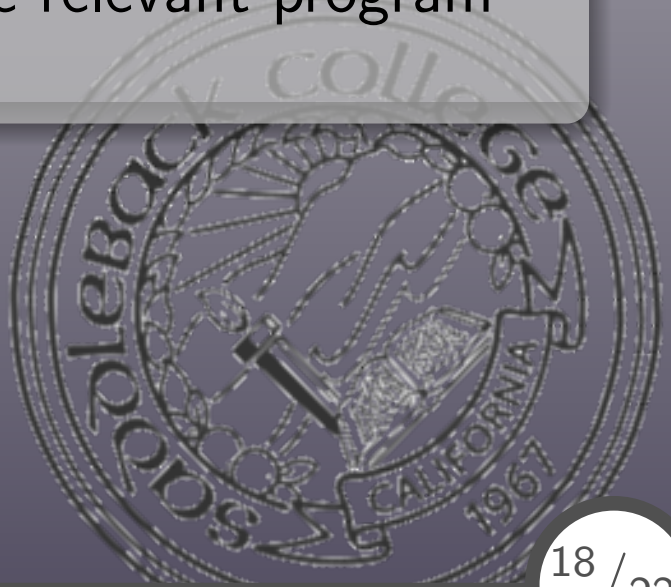
```
notice x changes:
 x: 1, y: 1
 x: 2, y: 1
 x: 3, y: 1
 x: 4, y: 1
 x: 5, y: 1
 x: 6, y: 1
 x: 7, y: 1
 x: 8, y: 1
 x: 9, y: 1
```

C++ Functions

Debugging

using drivers and stubs

- **drivers**: testing programs that exercises the relevant program
- **stubs**: testing functions that exercises the relevant program



C++ Functions

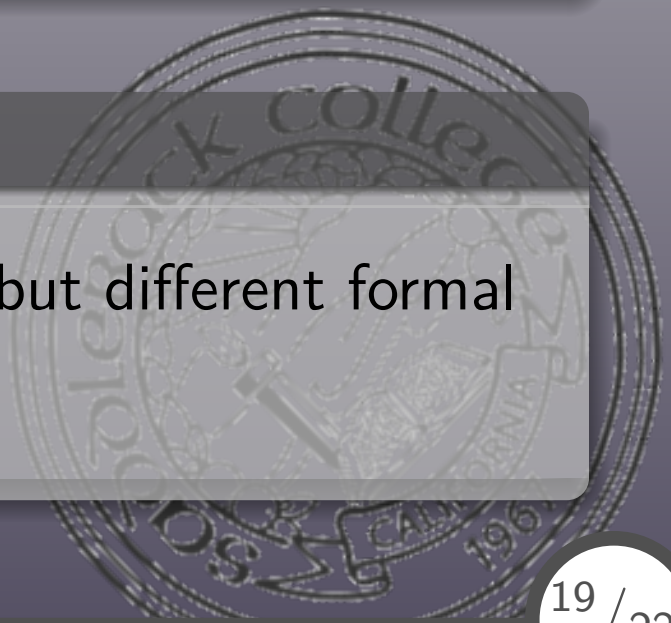
Function Overloading

functions have different formal parameter lists when

- they have a different number of formal parameters, or when...
- the data types of the formal parameters, in the order listed, differ in at least one position

function overloading occurs

- when two functions have the same name but different formal parameter lists



C++ Functions

Function Overloading

example

rather than ...

```
int lagerInt(int x, int y);
char largerChar(char a, char b);
double largerDouble(double u, double v);
string largerStr(string c, string d);
```

what about ...

```
int larger(int x, int y);
char larger(char a, char b);
double larger(double u, double v);
string larger(string c, string d);
```

and call like so:

```
larger(3, 5);
larger('A', 'E');
```

more to come when we get to object oriented programming...

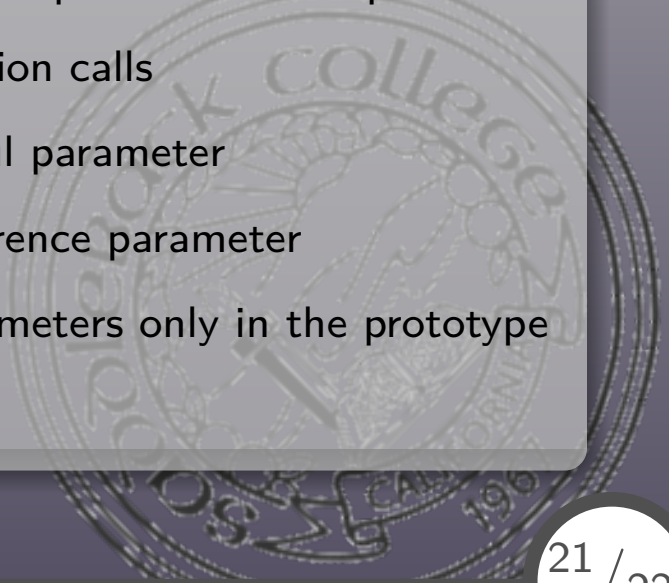
C++ Functions

Functions and Default Parameters

functions can have default values for parameters

```
void functionName(type1 var1, type2 var2=value, ...);
```

- if no value is specified for a parameter, the default value is used instead
- all of the default value parameters must be on the far right parameters of the function
- a function call should have no arguments to the right of unspecified default parameter
- default values can be constants, global variables, or function calls
- caller can specify a value other than default for any defaul parameter
- constant values cannot be used as default value to a reference parameter
- program with function prototypes must have default parameters only in the prototype



C++ Functions

Functions and Default Parameters

example

```
...
void defparFunction(int x, char a='A', int y=3);
...
int w,z;
z = 4;
w = 7;
char ch='K';
...
defparFunction(w, ch)   // ok
...
defparFunction(z, 'B', w)   // ok
...
defparFunction(w, z) // illegal; must omit z
...
```