Computer Science I

CS 135

2. Functions I: Passing by Value

René Doursat

Department of Computer Science & Engineering
University of Nevada, Reno

Fall 2005

Computer Science I

CS 135

0. Course Presentation
1. Introduction to Programming
2. **Functions I: Passing by Value**
3. File Input/Output
4. Predefined Functions
5. If and Switch Controls
6. While and For Loops
7. Functions II: Passing by Reference
8. 1-D and 2-D Arrays

Computer Science I

CS 135

2. Functions I: Passing by Value

a. **Breaking up a Program into Functions**

b. **Value-Returning & Void Functions**

c. **Code Layout: Declarations & Definitions**

d. **Scope: Local & Global Variables**

Computer Science I

CS 135

2. Functions I: Passing by Value

a. Breaking up a Program into Functions

- ✓ Example: a time-converter program
- ✓ Why functions are a good thing

b. Value-Returning & Void Functions

c. Code Layout: Declarations & Definitions

d. Scope: Local & Global Variables

2.a Breaking up a Program into Functions

Example: a time-converter program

- Problem: calculate total number of seconds
  - ✓ Write a C++ program that
    - prompts the user for some duration or elapsed time in hours, minutes and seconds
    - displays the same amount of time in total number of seconds

```
Enter time in hours, minutes and seconds:
2 40 37
The result is 9637 seconds
```

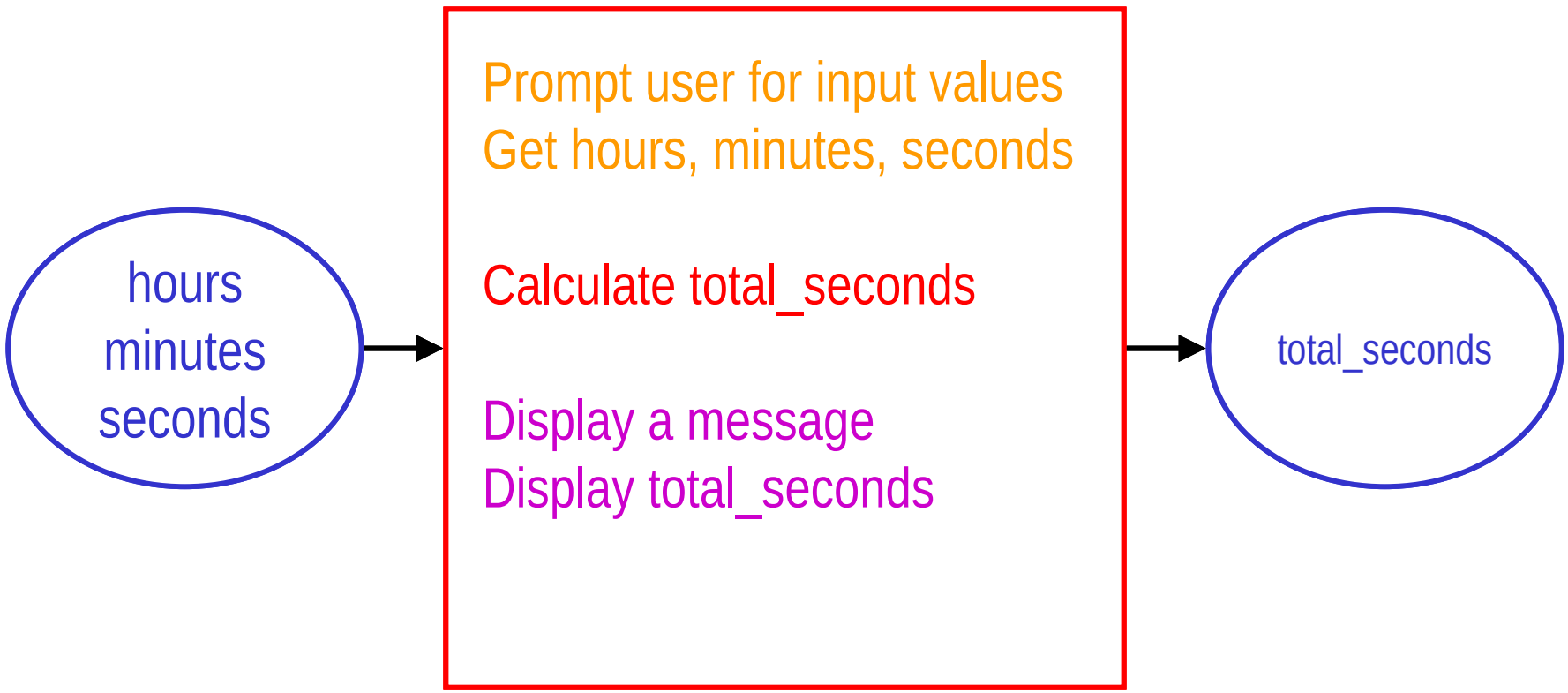
```
Enter time in hours, minutes and seconds:
0 2 5
The result is 125 seconds
```

Examples of output from the time-converter program

2.a Breaking up a Program into Functions

Example: a time-converter program

- Pseudocode
  - ✓ as usual, the 3 major steps are: prompt, calculate and display



2.a Breaking up a Program into Functions

Example: a time-converter program

- Official defining diagram

Input	Processing	Output
hours minutes seconds	Prompt user for input values Get hours, minutes, seconds Calculate total_seconds Display a message Display total_seconds	total_seconds

2.a Breaking up a Program into Functions

Example: a time-converter program

- Let's be more specific in the calculation part

Input	Processing	Output
hours minutes seconds	Prompt user for input values Get hours, minutes, seconds mins = hours*60 + minutes total_seconds = mins*60 + seconds Display a message Display total_seconds	total_seconds

2.a Breaking up a Program into Functions

Example: a time-converter program

```
#include <iostream>
using namespace std;

void main()
{
  // declare variables
  int hours, minutes, seconds;
  int mins, total_seconds;

  // prompt user for input values
  cout << "Enter time ";
  cin >> hours >> minutes >> seconds;

  // calculate number of seconds
  mins = hours*60 + minutes;
  total_seconds = mins*60 + seconds;

  // display result
  cout << "The result is ";
  cout << total_seconds << endl;
}
```

C++ code for the time-converter program

2.a Breaking up a Program into Functions

Example: a time-converter program

```
#include <iostream>
using namespace std;

void main()
{
  // declare variables
  int hours, minutes, seconds;
  int mins, total_seconds;

  // prompt user for input values
  cout << "Enter time ";
  cin >> hours >> minutes >> seconds;

  // calculate number of seconds
  total_seconds = calc_total_seconds(hours,
    minutes, seconds);

  // display result
  cout << "The result is ";
  cout << total_seconds << endl;
}
```

```
int calc_total_seconds(int h,
                       int m,
                       int s)
{
  int mins, total;

  mins = h*60 + m;
  total = mins*60 + s;

  return total;
}
```

C++ code for the time-converter program using a value-returning function

2.a Breaking up a Program into Functions

Example: a time-converter program

```
#include <iostream>
using namespace std;

void main()
{
  // declare variables
  int hours, minutes, seconds;
  int mins, total_seconds;

  // prompt user for input values
  cout << "Enter time ";
  cin >> hours >> minutes >> seconds;

  // calculate number of seconds
  total_seconds = calc_total_seconds(hours,
    minutes, seconds);

  // display result
  display_result(total_seconds);
}
```

```
int calc_total_seconds(int h,
                       int m,
                       int s)
{
  int mins, total;

  mins = h*60 + m;
  total = mins*60 + s;

  return total;
}

void display_result(int total)
{
  cout << "The result is ";
  cout << total << endl;
}
```

C++ code for the time-converter program using a value-returning function and a void function

2.a Breaking up a Program into Functions

Why functions are a good thing

- Write functions in your programs
  - ✓ keep the main steps of the program separated
  - ✓ instead of stuffing the whole sequence of steps into one big **`main`** function, try to split it into several functions
  - ✓ functions are like “building blocks” or “modules”
- Benefits of functions
  1. functions can be reused
  2. functions hide implementation details
  3. functions make team work possible
  4. functions make the code easier to understand

2.a Breaking up a Program into Functions

Why functions are a good thing

1. Functions can be reused

- ✓ once a block of code has been written in a function, it can be executed multiple times throughout a program
- ✓ when you need it, just call it in one line
- ✓ the same function can also be used in other programs without having to copy the block of code; again, just call it

```
void main()
{
  display_surface(7, 5);
  display_surface(100, 100);
  display_surface(4.6, 1.1);
}
```

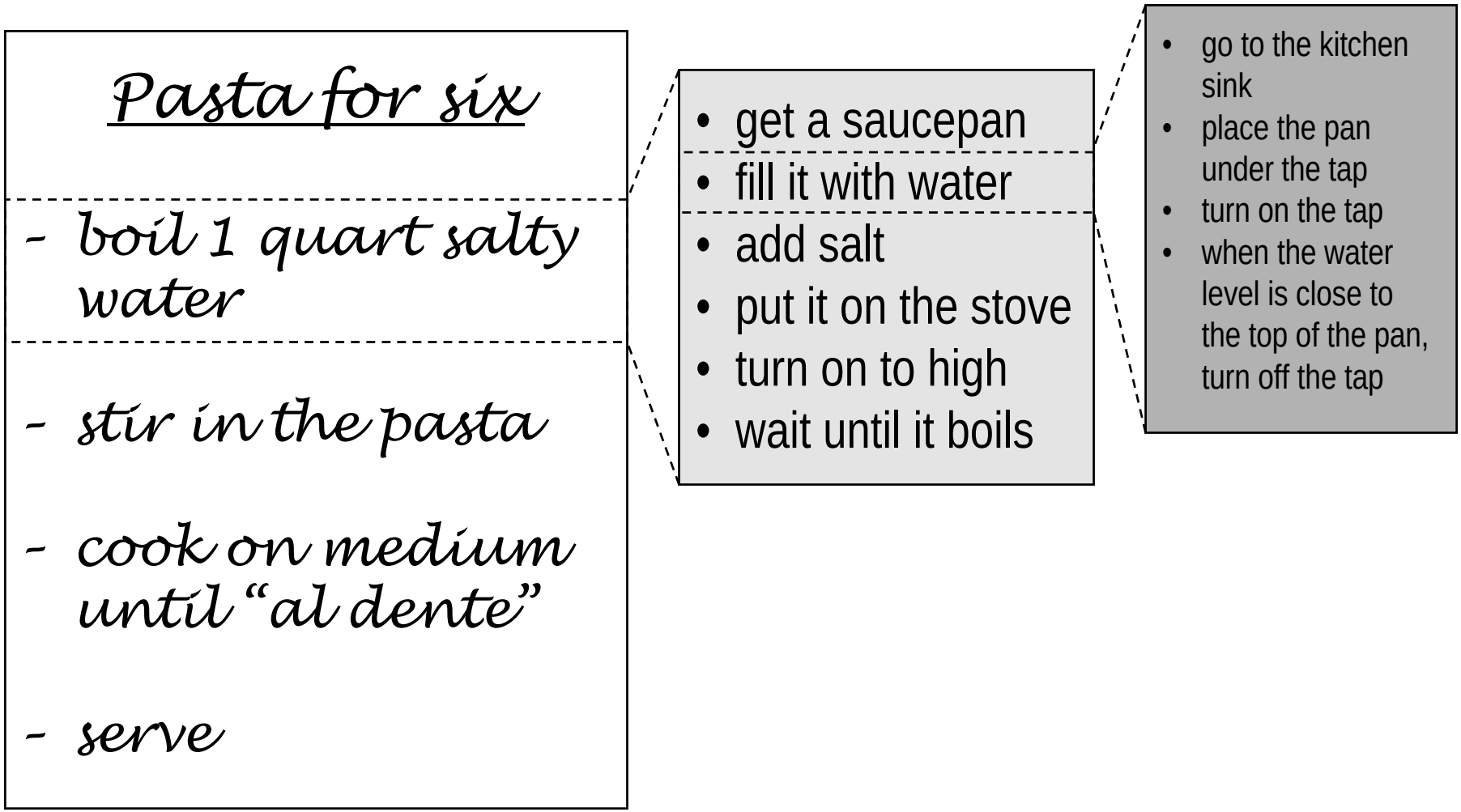
```
The surface is: 35
The surface is: 10000
The surface is: 5.06
```

```
void display_surface(double length,
                     double width)
{
  double surface = length * width;

  cout << "The surface is: ";
  cout << surface << endl;
}
```

2.a Breaking up a Program into Functions

Why functions are a good thing



2.a Breaking up a Program into Functions

Why functions are a good thing

2. Functions hide implementation details

- ✓ functions are like building blocks, they allow complicated programs to be divided into manageable pieces
- ✓ the microscopic details of how one block of code is written do not need to be visible at the higher level of the program
- ✓ this is called **encapsulation**

```
void main()
{
  boil_water();
  stir_in_pasta();
  cook();
  serve();
}
```

```
void boil_water()
{
  s = get_saucepan();
  fill_water(s);
  add_salt(s);
  put_on_stove(s);
  turn_stove_to_high();
  while (!boiling)
    wait();
}
```

```
void fill_water(int s)
{
  go_to_sink();
  place_under_tap(s);
  turn_on_tap();
  while (water_level< h)
    wait();
  turn_off_tap();
}
```

2.a Breaking up a Program into Functions

Why functions are a good thing

2. Functions hide implementation details (cont'd)

- ✓ the arguments to the function and the return data type define the "interface" between the function and the rest of the code
- ✓ as long as this interface doesn't change, the implementation can be rewritten without affecting the rest of the program

```
void main()
{
  display_surface(7, 5);
  display_surface(100, 100);
  display_surface(4.6, 1.1);
}
```

```
The surface is: 35
The surface is: 10000
The surface is: 5.06
```

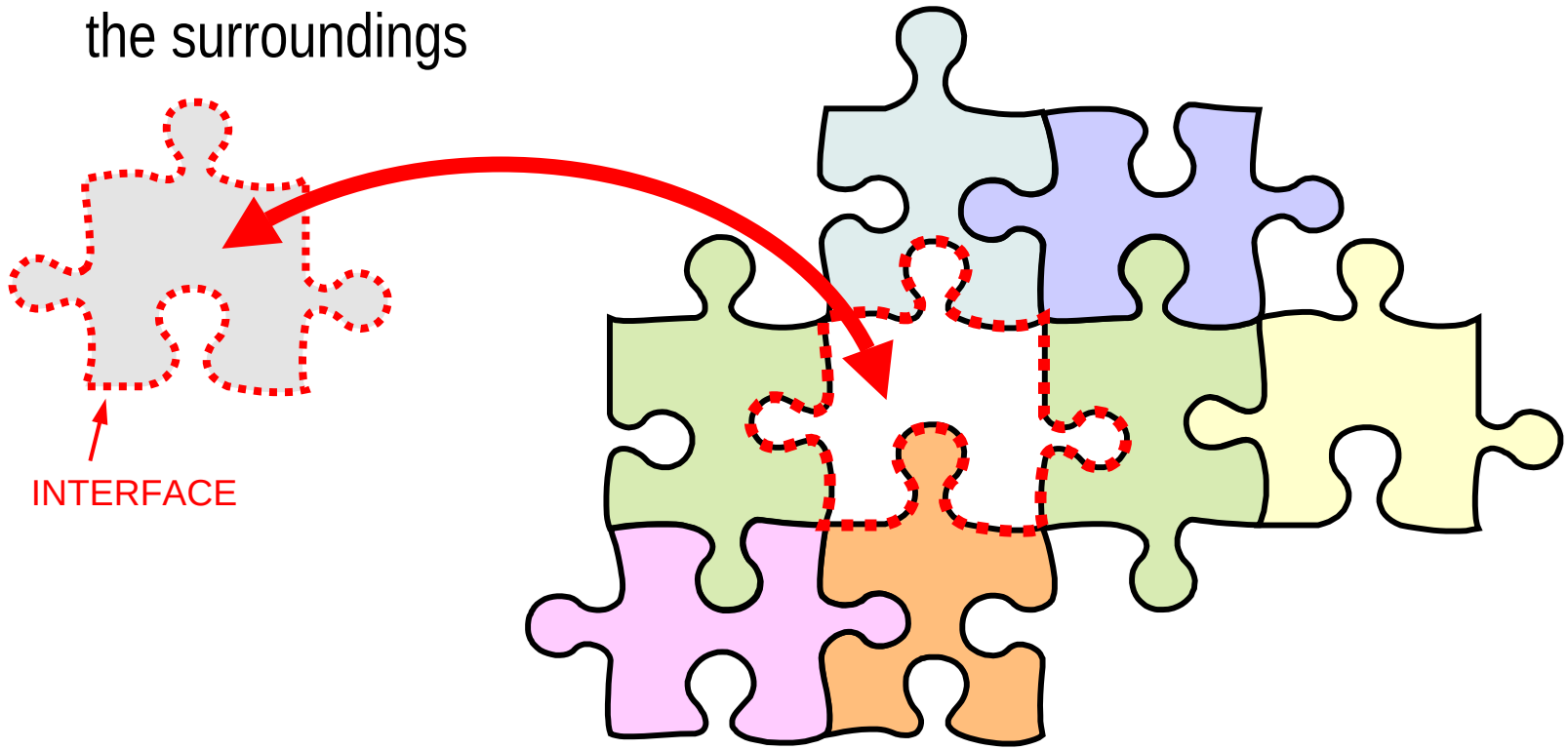
```
void display_surface(double length,
                     double width)
{
  cout << "The surface is: ";
  cout << (width * length) << endl;
}
```

2.a Breaking up a Program into Functions

Why functions are a good thing

2. Functions hide implementation details (cont'd)

- ✓ a programmer can focus on one part and construct it, debug it
- ✓ functions allow part change or replacement without impacting the surroundings



2.a Breaking up a Program into Functions

Why functions are a good thing

3. Functions make team work possible

- ✓ the work of writing a big code can be split among multiple programmers
- ✓ separate groups of programmers can work on separate modules
- ✓ each module can be thoroughly tested for correctness before it is incorporated into the final program.
- ✓ the final program is assembled from these building blocks

2.a Breaking up a Program into Functions

Why functions are a good thing

4. Functions make the code easier to understand

- ✓ breaking up the program up into chunks makes the code easier to read and the intent of the programmer easier to follow
- ✓ therefore, the code is easier to develop, debug, fix, maintain, upgrade, etc., whether by the same programmer (possibly months later) or by other people (possibly years later)

```
void main()
{
  boil_water();

  stir_in_pasta();

  cook();

  serve();
}
```

which code is easier to follow: a nice breakup into functional blocks? or . . .

```
void main()
{
   s = get_saucepan();
   go_to_sink();
   place_under_tap(s);
   turn_on_tap();
   while (water_level< h)
      wait();
   turn_off_tap();
   get_pasta();
   open_bag();
   throw_pasta_in(s);
   stir();
   ......
```

. . . one big ugly function with all the small steps crammed into it?

Computer Science I

CS 135

2. Functions I: Passing by Value

a. Breaking up a Program into Functions

- ✓ Example: a time-converter program
- ✓ Why functions are a good thing

b. Value-Returning & Void Functions

c. Code Layout: Declarations & Definitions

d. Scope: Local & Global Variables

Computer Science I

CS 135

2. Functions I: Passing by Value

a. Breaking up a Program into Functions

b. Value-Returning & Void Functions

- ✓ Value-returning functions with arguments
  - Math functions
- ✓ Value-returning functions without arguments
- ✓ Void functions with arguments
- ✓ Void functions without arguments

c. Code Layout: Declarations & Definitions

d. Scope: Local & Global Variables

2.b Value-Returning & Void Functions

Value-returning functions with arguments

- Functions that take arguments and return a value
  - ✓ example: functions calculating a formula

```
double calculate_surface(double length, double width)
{
    return length * width;
}
```

```
double calculate_average(int num1, int num2, int num3)
{
    return (num1 + num2 + num3)/3.0;
}
```

  - ✓ general syntax

arguments

return data type

```
<output data type> <function name>(<input data type> <variable>, ...)
{
    ...
    return <output value>;
}
```

2.b Value-Returning & Void Functions

Value-returning functions with arguments – Math functions

- Other example: the predefined math functions

Function	Standard Header File	Purpose	Parameter(s) Type	Result
`abs(x)`	`<cstdlib>`	Returns the absolute value of its argument: `abs(-7) = 7`	`int`	`int`
`ceil(x)`	`<cmath>`	Returns the smallest whole number that is not less than x: `ceil(56.34) = 57.0`	`double`	`double`
`cos(x)`	`<cmath>`	Returns the cosine of angle x: `cos(0.0) = 1.0`	`double` (radians)	`double`
`exp(x)`	`<cmath>`	Returns e^x, where e = 2.718: `exp(1.0) = 2.71828`	`double`	`double`
`fabs(x)`	`<cmath>`	Returns the absolute value of its argument: `fabs(-5.67) = 5.67`	`double`	`double`
`floor(x)`	`<cmath>`	Returns the largest whole number that is not greater than x: `floor(45.67) = 45.00`	`double`	`double`
`pow(x,y)`	`<cmath>`	Returns x^y; if x is negative, y must be a whole number: `pow(0.16, 0.5) = 0.4`	`double`	`double`

2.b Value-Returning & Void Functions

Value-returning functions without arguments

- Functions that take no arguments but return a value
  - ✓ example: functions prompting for a value

```
int prompt_for_age()
{
    int age;

    cout << "Please enter age: ";
    cin >> age;

    return age;
}
```

  - ✓ general syntax

no arguments

return data type

```
<output data type>  <function name>()
{
    ...
    return <output value>;
}
```

2.b Value-Returning & Void Functions

Void functions with arguments

- Functions that take arguments but return nothing
  - ✓ example: functions displaying a value

```
void display_age(int age)
{
    cout << "The age is: " << age << endl;
}
```

```
void display_surface(double length, double width)
{
    cout << "The surface is: " << (length*width) << endl;
}
```

  - ✓ general syntax

arguments

```
void <function name>(<input data type> <variable>, ...)
{
    ...
}
```

2.b Value-Returning & Void Functions

Void functions without arguments

- Functions that take no arguments and return nothing
  - ✓ example: **`main`** function or high-level steps in the **`main`**

```
void main()
{
    boil_water();
    stir_in_pasta();
    cook();  serve();
}
```

```
void boil_water()
{
    ...
```

  - ✓ general syntax

```
void <function name>()
{
    ...
}
```

Computer Science I

CS 135

2. Functions I: Passing by Value

a. Breaking up a Program into Functions

b. Value-Returning & Void Functions

- ✓ Value-returning functions with arguments
  - Math functions
- ✓ Value-returning functions without arguments
- ✓ Void functions with arguments
- ✓ Void functions without arguments

c. Code Layout: Declarations & Definitions

d. Scope: Local & Global Variables

Computer Science I

CS 135

2. Functions I: Passing by Value

a. Breaking up a Program into Functions

b. Value-Returning & Void Functions

c. Code Layout: Declarations & Definitions

- ✓ General syntax of a function definition
- ✓ General syntax of a function declaration or "prototype"
- ✓ General code layout

d. Scope: Local & Global Variables

2.c Code Layout: Declarations & Definitions

General syntax of a function definition

- To include a function in a code, you need to know the properties of the function
  - ✓ the **header** of the function, which includes
    - the name of the function
    - the number of parameters, if any
    - the data type of each parameter, if any
    - the data type of the value computed by the function, called the “type of the function”, if any
  - ✓ the **body** of the function, which contains
    - the code required to accomplish the task

2.c Code Layout: Declarations & Definitions

General syntax of a function definition

- Anatomy of a function definition

header →
body

```
double calculate_surface(double length, double width)
{
    ...
    return surface;
}
```

```
int prompt_for_age()
{
    ...
    return age;
}
```

```
void display_age(int age)
{
    ...
}
```

```
void main()
{
    ...
}
```

- function type
- function name
- argument list (types & names)
- { body }

2.c Code Layout: Declarations & Definitions

General syntax of a function declaration or “prototype”

- The function declaration or “prototype” is simply the header without body
  - ✓ remove the whole block between curly braces
  - ✓ add a semi-colon
  - ✓ you can also remove the input variable names and leave only the input data types

2.c Code Layout: Declarations & Definitions

General syntax of a function declaration or "prototype"

- Anatomy of a function declaration

header →

```
double calculate_surface(double, double);
```

NO body

```
int prompt_for_age();
```

```
void display_age(int);
```

```
void main();
```

- function type
- function name
- argument list (types ~~& names~~)
- ~~{ body }~~

2.c Code Layout: Declarations & Definitions

General code layout

- Functions show up in three different places in the code
  1. functions are **declared**
     - the prototype is just the header with a semi-colon
     - it is generally located at the beginning of the program, in any case always before the function is used and defined
  2. functions are **called** from inside the main or another function
     - the function call can be placed anywhere within the body of another function
  3. functions are **defined**
     - the definition is made of a copy of the function header (with argument names) and the body of the function between curly braces **{}** that contains the actual implementation

2.c Code Layout: Declarations & Definitions

General code layout

```
...
int calc_total_seconds(int, int, int);
void display_result(int);
```

1. functions are declared

```
void main()
{
  ...
  total_seconds = calc_total_seconds(hours,
    minutes, seconds);
  ...
  display_result(total_seconds);
}
```

2. functions are called inside the body of other functions

```
int calc_total_seconds(int h, int m, int s)
{
  ...
}
```

```
void display_result(int total)
{
  ...
}
```

3. functions are defined

Example of code layout with functions

2.c Code Layout: Declarations & Definitions

General code layout

```
...
int calc_total_seconds(int, int, int);
void display_result(int);
```

```
void main()
{
  ...
  total_seconds = calc_total_seconds(hours,
    minutes, seconds);
  ...
  display_result(total_seconds);
}
```

```
int calc_total_seconds(int h, int m, int s)
{
  ...
}
```

```
void display_result(int total)
{
  ...
}
```

when declaring a function, argument names are not necessary

when calling a function, just pass the variable you are using without the data type

when defining a function, you must give names to the arguments, but preferably different from the passed variable

Example of code layout with functions

Computer Science I

CS 135

2. Functions I: Passing by Value

a. Breaking up a Program into Functions

b. Value-Returning & Void Functions

c. Code Layout: Declarations & Definitions

- ✓ General syntax of a function definition
- ✓ General syntax of a function declaration or "prototype"
- ✓ General code layout

d. Scope: Local & Global Variables

Computer Science I

CS 135

2. Functions I: Passing by Value

a. Breaking up a Program into Functions

b. Value-Returning & Void Functions

c. Code Layout: Declarations & Definitions

d. Scope: Local & Global Variables

- ✓ Definition of scope
- ✓ Local variables
- ✓ Global variables

2.d Scope: Local & Global Variables

Definition of scope

- Scope of a variable
  - ✓ the **scope** is the portion of a program in which the variable has been defined
  - ✓ the scope of **local** variables is the block between curly braces **{** and **}** in which the variable was defined, for example:
    - the variables declared inside the main
    - the variables declared inside another function
    - the arguments in the header of the function
  - ✓ the scope of **global** variables is the whole program
    - these variables are declared outside of any function, generally at the top of the program
    - however, they should be avoided (they can cause conflicts)

2.d Scope: Local & Global Variables

Local variables

```
#include <iostream>
using namespace std;

void main()
{
  // declare variables
  int hours, minutes, seconds;
  int total_seconds;

  // prompt user for input values
  cout << "Enter time ";
  cin >> hours >> minutes >> seconds;

  // calculate number of seconds
  total_seconds = calc_total_seconds(hours,
    minutes, seconds);

  // display result
  cout << "The result is ";
  cout << total_seconds << endl;
}
```

```
int calc_total_seconds(int h,
                       int m,
                       int s)
{
  int mins, total;

  mins = h*60 + m;
  total = mins*60 + s;

  return total;
}
```

these 2 variables and 3 arguments are only local to the calc function; they are not known by the main

these 4 variables are local to the main function and not known anywhere else

Example of local variables

2.d Scope: Local & Global Variables

Global variables

```
#include <iostream>
using namespace std;

const int mins_per_hr = 60;

void main()
{
  // declare variables
  int hours, minutes, seconds;
  int total_seconds;

  // prompt user for input values
  cout << "Enter time ";
  cin >> hours >> minutes >> seconds;

  // calculate number of seconds
  total_seconds = calc_total_seconds(hours,
    minutes, seconds);

  // display result
  cout << "The result is ";
  cout << total_seconds << endl;
}
```

```
int calc_total_seconds(int h,
                       int m,
                       int s)
{
  int mins, total;

  mins = h*mins_per_hr + m;
  total = mins*mins_per_hr + s;

  return total;
}
```

this variable is declared outside of any function and can be reused anywhere in the code, like here;
→ it is called a global variable

but now you know that global variables exist . . . don't use them!

Example of global variables

Computer Science I

CS 135

2. Functions I: Passing by Value

a. Breaking up a Program into Functions

b. Value-Returning & Void Functions

c. Code Layout: Declarations & Definitions

d. Scope: Local & Global Variables

- ✓ Definition of scope
- ✓ Local variables
- ✓ Global variables

Computer Science I

CS 135

2. Functions I: Passing by Value

a. Breaking up a Program into Functions

b. Value-Returning & Void Functions

c. Code Layout: Declarations & Definitions

d. Scope: Local & Global Variables

Computer Science I

CS 135

0. Course Presentation
1. Introduction to Programming
2. Functions I: Passing by Value
3. **File Input/Output**
4. **Predefined Functions**
5. **If and Switch Controls**
6. **While and For Loops**
7. **Functions II: Passing by Reference**
8. **1-D and 2-D Arrays**