Call-by-Reference Functions
Procedural Abstractions
Numerical Conversions

CS 16: Solving Problems with Computers I
Lecture #9

Ziad Matni
Dept. of Computer Science, UCSB

Announcements

- **Homework #8 due today**
- **Lab #5 is due on Friday at Noon**
- Homework Solutions are now online at:

http://cs.ucsb.edu/~zmatni/cs16/hwSolutions/

- Your grades are NOT ON GAUCHOSPACE anymore.
  Instead go to:

http://cs.ucsb.edu/~zmatni/cs16/CS16Grades_Fa2016.htm

Lecture Outline

- Call-By-Reference Parameters
- Using Procedural Abstraction

- Binary Arithmetic 1:
  - Conversions

Call-by-Value vs Call-by-Reference

- When you call a function, your arguments are getting passed on as *values*
  - At least, with what we've seen so far…
  - The call `func(a, b)` passes on the *values* of **a** and **b**
- You can also call a function with your arguments used as *references* to the actual variable location in memory
  - Why would we want to do that?

Call-by-Reference Parameters

- "Call-by-reference" parameters allow us to change the **variable** used in the function call

- Arguments for call-by-reference parameters must be variables, not numbers
  - i.e. fn(var), not fn(5)

- We use the ampersand symbol (**&**) to distinguish a variable as being called-by-reference, in a function definition

Call-by-Reference Parameters

- Recall that, up until now, we have changed the values of formal parameters in a function body, **but we have not changed the arguments found in the function call!**
  - Example: when you call **func(a, b, c)**, you might get a returned value for **func**, but **a**, **b**, and **c** do not change after the call.

- So if you want to get an input (via **`cin`**) using a function using call-by-value, it wouldn’t work!

Call-by-Reference Example

```
void get_input(double& f_variable)
 {
        using namespace std;
        cout << “ Convert a Fahrenheit temperature”
             << “ to Celsius.\n”
              << “ Enter a temperature in Fahrenheit: “;
    cin >> f_variable;
}
```

- ‘**&**’ symbol (ampersand) identifies **f_variable** as a call-by-reference parameter
  - Has to be used in both declaration and definition!

Call-By-Reference Details

- Call-by-reference works almost as if the argument ***variable itself*** is substituted for the formal parameter, not the argument's ***value***

- In reality, it's the ***memory location of the argument variable*** that is given to the formal parameter
  - Whatever is done to a formal parameter in the function body, is ***actually done to the value at the memory location of the argument variable***

Call-by-Reference Behavior

- Assume variables **`first`** and **`second`** are assigned memory addresses **1010** and **1012** by the compiler, respectively

- Now a function call executes: **`get_numbers(first, second);`**
- The function is defined as:

```
    void get_numbers(int& first, int& second) {
        cout << “Enter two integers: ”
        cin >> first >> second; }
```

- The function may as well say:

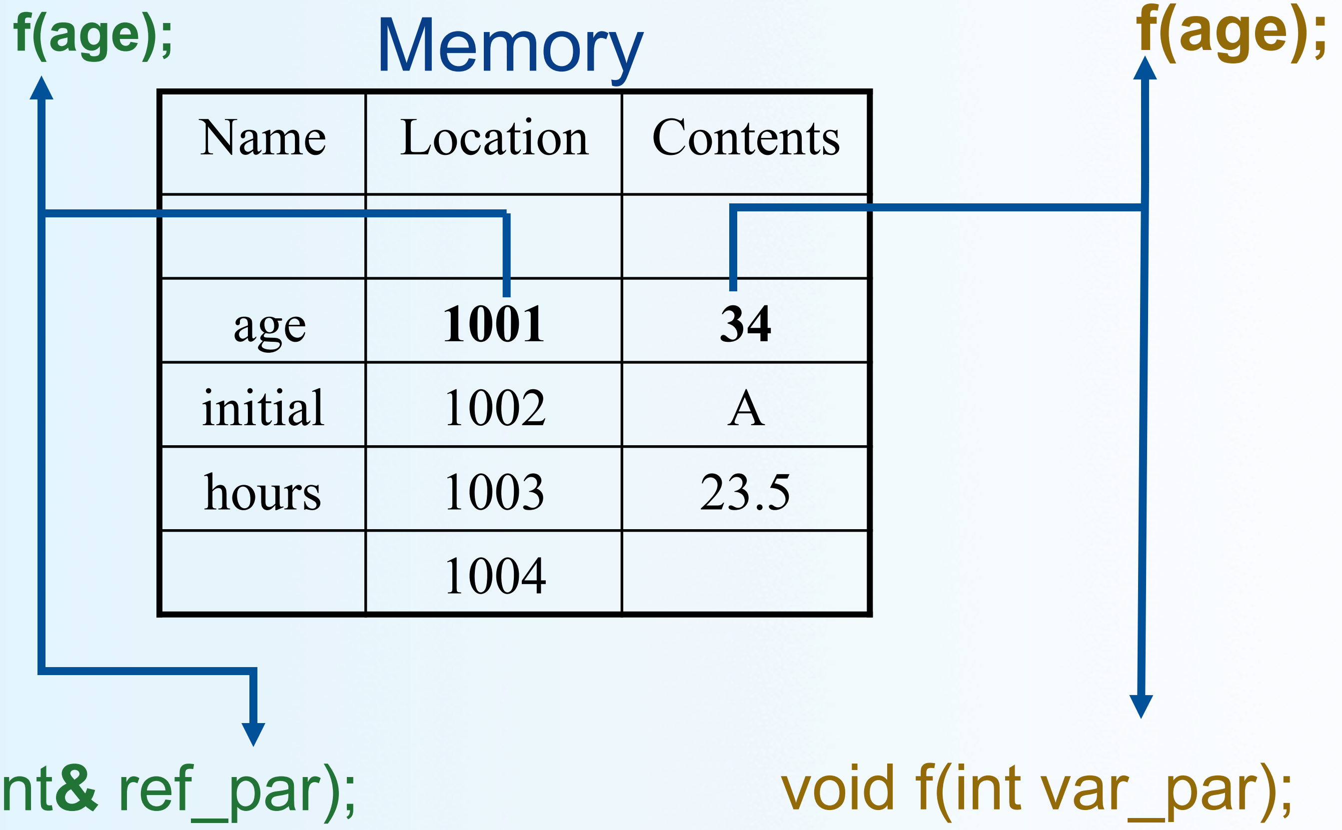
```
    void get_numbers(the int variable at memory location 1010,
the int variable at memory location 1012) {
        cout << “Enter two integers: ”
        cin >> the variable at memory location 1010;
            >> the variable at memory location 1012; }
```

Call-by-Reference Behavior

- After the function is called, the values of **`first`** and **`second`** are changed – not just in the function, but in actual memory
- The values of **`first`** and **`second`** are in the calling block

Call by-Reference vs by-Value

- **Call-by-reference**
  - The function call:
- **Call-by-value**
  - The function call:



f(age);

Memory

Name	Location	Contents
age	**1001**	**34**
initial	1002	A
hours	1003	23.5
	1004	

f(age);

void f(int& ref_par);

void f(int var_par);

Example: swap_values

```
void swap(int& variable1, int& variable2)
{
     int temp = variable1;
     variable1 = variable2;
     variable2 = temp;
}
```

- If called with **`swap(first_num, second_num);`**
  - **`first_num`** is substituted for variable1 in the parameter list
  - **`second_num`** is substituted for variable2 in the parameter list
  - temp is assigned the value of variable1 (**`first_num`**) since the next line will loose the value in **`first_num`**
  - variable1 (**`first_num`**) is assigned the value in variable2 (**`second_num`**)
  - variable2 (**`second_num`**) is assigned the original value of variable1 (**`first_num`**) which was stored in temp

Mixed Parameter Lists

- Call-by-value and call-by-reference parameters can be mixed in the same function

- Example:
  `void good_stuff(int& par1, int par2, double& par3);`

  - par1 and par3 are call-by-reference formal parameters
    - Changes in par1 and par3 change the argument variable

  - par2 is a call-by-value formal parameter
    - Changes in par2 do not change the argument variable

Choosing Parameter Types

- How do you decide whether a call-by-reference or call-by-value formal parameter is needed?
- Does the function need to change the value of the variable used as an argument?
  - Yes? Use a call-by-reference formal parameter
  - No? Use a call-by-value formal parameter

Caution!
Inadvertent Local Variables

- Forgetting the ampersand (&) creates a call-by-value parameter
  - The value of the variable will not be changed
- The formal parameter is a local variable that has no effect outside the function
  - Hard error to find…it looks right!

Using Procedural Abstraction

- Functions should be designed so they can be used as black boxes

- To use a function, the declaration and comment should be sufficient

- Programmer should not need to know the details of the function to use it

Functions Calling Functions

- A function body may contain a call to another function
- The called function declaration must still appear before it is called
- Functions **cannot be defined** in the body of another function

```
void order (int&, int&);
void swap_values (int&, int&);


int main () {
…
…
    order (a, b);
…
…
    return 0;   }


void order(int&  n1, int&  n2) {
    if (n1 > n2)
    swap_values(n1, n2);   }


void swap_values(int& n1, int& n2) {
    int temp = n2;
    n2 = n1;
    n1 = temp;  }
```

Pre- and Post-Conditions

Concepts of pre-condition and post-condition

Pre-condition

- States what is assumed to be true when the function is called
- Function should not be used unless the precondition holds

Post-condition

- Describes the effect of the function call
- Tells what will be true after the function is executed (when the precondition holds)
- If the function returns a value, that value is described
- Changes to call-by-reference parameters are described

swap_values revisited

- Using preconditions and postconditions the declaration of **swap_values** becomes:

```
void swap_values(int& n1, int& n2);
//Precondition:  variable1 and variable2 have
//               been given values
// Postcondition:  The values of variable1 and
//                 variable2 have been interchanged
```

Function Celsius

- Preconditions and post-conditions make the declaration for celsius():

```
double celsius(double farenheit);
//Precondition:  fahrenheit is a temperature
//               expressed in degrees Fahrenheit
//Postcondition: Returns the equivalent temperature
//               expressed in degrees Celsius
```

Why use Pre- and Post-conditions?

- Pre-conditions and post-conditions should be the first step in designing a function

- Specify what a function should do:
  - Always specify what a function should do before designing how the function will do it
  - This minimizes design errors and time wasted writing code that doesn't match the task at hand

Case Study: Supermarket Pricing

Problem definition

- Determine the retail price of an item, given a suitable input
  - 5% markup if item should sell in a week
  - 10% markup if item expected to take more than a week
  - 5% for up to 7 days, changes to 10% at 8 days

Primary analysis

- Input:
  - The wholesale price and the estimate of days until item sells (turnover)
- Output:
  - The retail price of the item

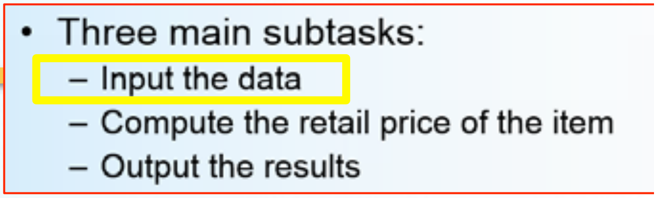
- **What subtasks do you think we need here?**

Supermarket Pricing: Problem Analysis

- Three main subtasks:
  - Input the data
  - Compute the retail price of the item
  - Output the results

- Each task can be implemented with a function
  - Note the use of **call-by-value** and **call-by-reference** parameters in the following function declarations (next slide)

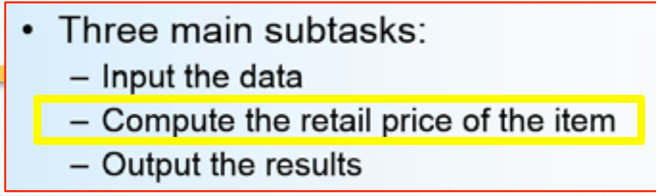
Supermarket Pricing: Function **get_input**



- Three main subtasks:
  - Input the data
  - Compute the retail price of the item
  - Output the results

```
void get_input(double& cost, int& turnover);
//Precondition: User is ready to enter values correctly.
//Postcondition: The value of cost has been set to
//               the wholesale cost of one item.
//
//                The value of turnover has been
//               set to the expected number of
//               days until the item is sold.
```

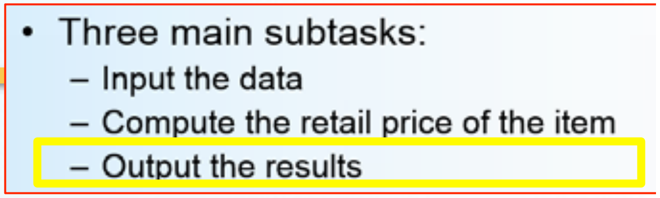
Supermarket Pricing: Function **price**



- Three main subtasks:
  - Input the data
  - Compute the retail price of the item
  - Output the results

```
double price(double cost, int turnover);
//Precondition: cost is the wholesale cost of one item
//              turnover is the expected
//              number of days until the item is sold.
//
//Postcondition: returns the retail price of the item
```

Supermarket Pricing: Function **give_output**



- Three main subtasks:
  - Input the data
  - Compute the retail price of the item
  - Output the results

```
void give_output(double cost, int turnover, double price);
//Precondition:  cost is the wholesale cost of one item;
//               turnover is the expected time until sale
//               of the item;
//               price is the retail price of the item.
//
//Postcondition: The values of cost, turnover, and price
//                are written to the screen.
```

Supermarket Pricing: **main()** function

- With the functions declared, we can write the main function:

```
int main()
  {
    double wholesale_cost, retail_price;
    int shelf_time;

    get_input(wholesale_cost, shelf_time);

    retail_price = price(wholesale_cost, shelf_time);

    give_output(wholesale_cost, shelf_time, retail_price);

    return 0;
   }
```

Supermarket Pricing: Algorithm Design of **price** function

- Implementations of **get_input** and **give_output** are straightforward, so we'll concentrate on the price function

- pseudocode for the price function:

  ```
  If turnover <= 7 days then
          return (cost + 5% of cost);
  else
          return (cost + 10% of cost);
  ```

Supermarket Pricing: Constants for the **price** function

- The mark-up values and the threshold number won't change, so we can represent them with constants:

```
const double LOW_MARKUP = 0.05;  // 5%
const double HIGH_MARKUP = 0.10; // 10%
const int THRESHOLD = 7;   // At 8 days use HIGH_MARKUP
```

Supermarket Pricing: Coding the **price** function

- The body of the price function

```
{
     if (turnover <= THRESHOLD)
         return ( cost + (LOW_MARKUP * cost) ) ;
     else
         return ( cost + ( HIGH_MARKUP * cost) ) ;
}
```

- See the complete program in the textbook, Chapter 5, pages 279-280

Supermarket Pricing : Program Testing

- What are some testing strategies that we can use?
  - TEST COVERAGE: Covering all possible conditions of inputs and checking the outputs against expected results.

- Use data that tests both the high and low markup cases
  - Test boundary conditions, where the program is expected to change behavior or make a choice
  - In function **price**, 7 days is a boundary condition
  - Test for exactly 7 days as well as one day more and one day less

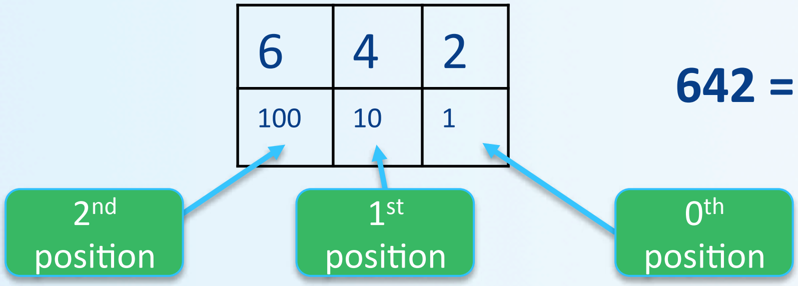
Counting Numbers in Different Bases

- We "normally" count in 10s
  - Base 10: decimal numbers
  - Number symbols are 0 thru 9
- Computers count in 2s
  - Base 2: binary numbers
  - Number symbols are 0 and 1
  - Represented with **1 bit** ($2^1 = 2$)
- Other convenient bases in computer architecture:
  - Base 8: octal numbers
  - Number symbols are 0 thru 7
  - Represented with **3 bits** ($2^3 = 8$)
  - Base 16: hexadecimal numbers
  - Number symbols are 0 thru F
    - A = 10, B = 11, C = 12, D = 13, E = 14, F = 15
  - Represented with **4 bits** ($2^4 = 16$)
  - **Why are 4 bit representations convenient???**

Natural Numbers

Counting **642** as 600 + 40 + 2
is counting in TENS (aka BASE 10)

There are 6 HUNDREDS
There are 4 TENS
There are 2 ONES



6	4	2
100	10	1

2nd position

1st position

0th position

642 = 600 + 40 + 2

Positional Notation in Decimal

Continuing with our example…

642 in base 10 *positional notation* is:

$$
\begin{aligned}
& \,6 \times 10^2 = 6 \times 100 = 600 \\
& + 4 \times 10^1 = 4 \times 10 = 40 \\
& + 2 \times 10^0 = 2 \times 1 = 2 = 642 \text{ in base } 10
\end{aligned}
$$

Positional Notation

Anything → DEC

What if "642" is expressed in the base of 13?

$$
\begin{aligned}
6 \times 13^2 &= 6 \times 169 &&= 1014 \\
+\,4 \times 13^1 &= 4 \times 13 &&= 52 \\
+\,2 \times 13^0 &= 2 \times 1 &&= 2 \\
&= 1068 \text{ in base } 10
\end{aligned}
$$

So, "642" in base 13 is equivalent to "1068" in base 10

BUT WHO COUNTS IN BASE 13???!?!?

Maybe, aliens with 13 fingers???

HUMANS LIKE TO COUNT IN BASE 10

We have 10 fingers…

Coincidence????

(probably the reason, but it’s historically unverified)

COMPUTERS ARE DIGITAL MACHINES

THEY ARE DESIGNED TO COUNT IN…

2

Positional Notation in Binary

11011 in base 2 *positional notation* is:

$$\begin{aligned} & 1 \times 2^4 = 1 \times 16 = 16 \\ + & 1 \times 2^3 = 1 \times 8 \quad = 8 \\ + & 1 \times 2^2 = 1 \times 4 \quad = 4 \\ + & 0 \times 2^1 = 1 \times 2 \quad = 0 \\ + & 1 \times 2^0 = 1 \times 1 \quad = 1 \end{aligned}$$

So, **1011** in base 2 is 16 + 8 + 0 + 2 + 1 = **27** in base 10

Converting Binary to Decimal

Q: What is the decimal equivalent of the binary number ***1101110****?*

A: Look for the position of the digits in the number.
This one has 7 digits, therefore positions 0 thru 6

1	**1**	**0**	**1**	**1**	**1**	**0**
64	32	16	8	4	2	1
2^6	2^5	2^4	2^3	2^2	2^1	2^0

1 x 2^6 = 1 x 64 = 64
+ 1 x 2^5 = 1 x 32 = 32
+ 0 x 2^4 = 0 x 16 = 0
+ 1 x 2^3 = 1 x 8 = 8
+ 1 x 2^2 = 1 x 4 = 4
+ 1 x 2^1 = 1 x 2 = 2
+ 0 x 2^0 = 0 x 1 = 0
= 110 in base 10

Converting Binary to Octal and Hexadecimal

(or any base that's a power of 2)

- Binary is 1 bit
- Octal is 3 bits
- Hexadecimal is 4 bits

- Use the "group the bits" technique
  - Always start from the *least significant digit*
  - Group every 3 bits together for bin → oct
  - Group every 4 bits together for bin → hex

Converting Binary to Octal and Hexadecimal

- Take the example: ***10100110***

…to octal:

1 0 | 1 0 0 | 1 1 0

2 ***4*** ***6***

246 in octal

…to hexadecimal:

1 0 1 0 | 0 1 1 0

10 ***6***

A6 in hexadecimal

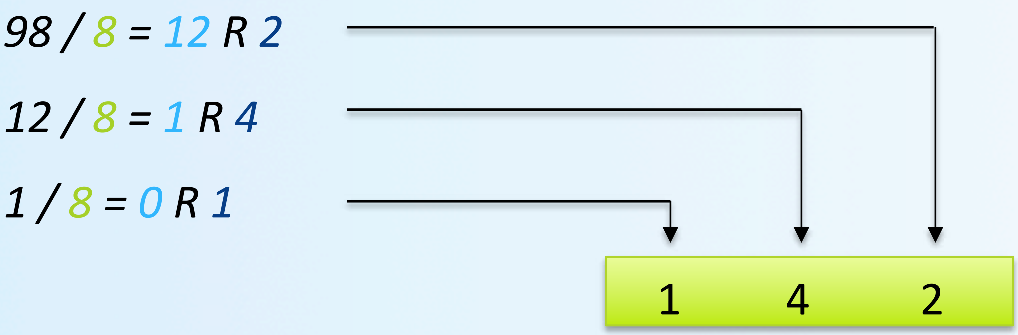
Converting Decimal to Other Bases

Algorithm for converting number in base 10 to other bases

While (the quotient is not zero)

1. Divide the decimal number by the new base
2. Make the remainder the next digit to the left in the answer
3. Replace the original decimal number with the quotient
4. Repeat until your quotient is zero

Example: What is 98 (base 10) in base 8?



98 / 8 = 12 R 2

12 / 8 = 1 R 4

1 / 8 = 0 R 1

Converting Decimal into Binary

Convert 54 (base 10) into binary and hex:

- 54 / 2 = 27 R 0
- 27 / 2 = 13 R 1
- 13 / 2 = 6 R 1
- 6 / 2 = 3 R 0
- 3 / 2 = 1 R 1
- 1 / 2 = 0 R 1

Sanity check:
110110
= 2 + 4 + 16 + 32
= 54

54 (decimal) = 110110 (binary)
= 36 (hex)

TO DOs

- Homework #9 due Thursday 10/27
- Lab #5
  - Due Friday, 10/28, at noon

</LECTURE>