# CISC 1600/1610 Computer Science I

Functions, continued

Professor Daniel Leeds
dleeds@fordham.edu
JMH 328A

## Variable scope

Variables declared in a function

- are **local** to that function
- are invisible to all other functions

`int main()` is a function



## Formal parameters

"Formal parameters" are the variables in the function head

```
float triple(float inNum)
{
   float tripledNum;
   tripledNum=3*inNum;
   return tripledNum;
}
```

Function head

Function body



## Formal parameters

- **Local** to the function
- Used as if they were declared in function body – **do not** re-declare in function body
- When function is called, parameters initialized to the values of the arguments in the function call

```
float triple(float inNum)
{
   float tripledNum;
   tripledNum=3*inNum;
   return tripledNum;
}
```



## Formal parameter names

- Parameter names do not have to match names of variables used in function call
- Different programmer can write `int main()` and functions used by `int main()`



## Broader scope: global variables

- Global variables visible to all functions
- Declared outside of all functions
- Must be declared prior to first use

```
#include<iostream>
using namespace std;
const float PI=3.14;
    // visible to main and to areaCircle

// compute area of circle
float areaCircle(float radius);

int main() { ...}
float areaCircle(float radius) {...}
```

## More on global variables

- Useful to define global constants
- Very risky to define non-constant global variables
  - try to keep track of what functions change the variable

7

## Function overloading

"Overloading" when multiple functions with same name but:

- different number of parameters
- different types of parameters

Compiler determines which function to use

8

## Overloaded averaging function

```
float average(int num1, int num2) {
   return (num1+num2)/2.0;
}

float average(int num1, int num2, int
num3) {
   return ???;
}
```

9

## `void` functions

- void function returns no value

Example definition:

```
void greetUser(string userName){
  cout << "Hello " << userName
       << endl;
  return;
}
```

Example call:

```
  greetUser(userName);
```

**NOT:** ~~`cout << greetUser(userName);`~~

10

## Use of `return;`

- In void funtion, can use `return;`
- When evaluated, `return;` terminates function

11

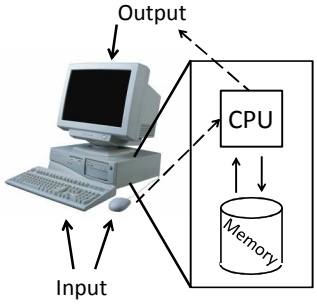
## Computer system structure

Central processing unit (CPU) – performs all the instructions

Memory – stores data and instructions for CPU

Input – collects information from the world

Output – provides information to the world




12

## The binary representation

- Each variable is represented by a certain number of 0s and 1s
- Each 0-or-1 is a bit
- 8 bits in a row is a byte

`int numStudents = 33;` assigns a binary code to memory: `00000000000000000000000000001100`

$$2^3x1 + 2^2x1 + 2^1x0 + 2^0x0$$
$$8x1 + 4x1 + 2x0 + 1x0$$
$$14$$

13

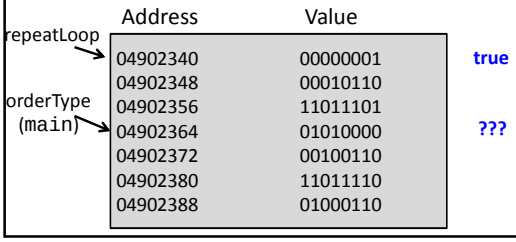
## Variable types, revisited

| | | |
|---|---|---|
| `char` | single character ('a', 'Q') | 1 byte |
| `int` | integers (-4, 82) | 4 bytes |
| `bool` | logic (true or false) | 1 byte |
| `float` | real numbers (1.3, -0.45) | 4 bytes |
| `string` | text ("Hello", "reload") | ? bytes |
| `vector` | sequence of values ({16,5}, {-2.3,3.4,-0.4}) | ? bytes |

14

## Variables – locations in memory

- Each variable indicates a location in memory
- Each location holds a value
- Value can change as program progresses

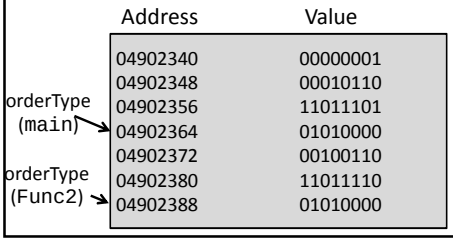



15

## Memory usage by functions

"Call-by-value":

- provide function with the value held in a variable input
- assign value to new internal variable

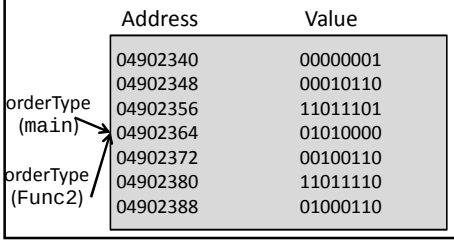



16

## Memory usage by functions

"Call-by-reference":

- provide function with the **address** of a variable input
- assign value into old address




17

## Call-by-Reference Syntax

- Use & to indicate a variable is called by reference
- Use & both in declaration and definition

```
void get_letters(char& letter1, char& letter2);

...

void get_letters(char& letter1, char& letter2)
{
   cout << "Enter two letters: ";
   cin >> letter1 >> letter2;
}
```

18

## Call-by-reference vs. Call-by-value

- Call-by-value preserves the value of the original input argument
- Call-by-reference can change the value of the original input argument
  - Effectively allows return of multiple values from function

19

```
int mysteryFunc(int& num1);

int main() {
  int a=5;
  cout << mysteryFunc(a) << endl;
  cout << a << endl;
  return 0;
}

int mysteryFunc(int &num1) {
  num1 += 3;
  return num1/4;
}
```

What does this do?

20

```
int mysteryFunc2(int inNum);

int main() {
  int a=3;
  cout << mysteryFunc2(a);
  cout << a;
  return 0;
}

int mysteryFunc2(int inNum) {
  inNum = inNum*inNum;
  return inNum;
}
```

What does this do?

21

## Call-by-reference: Input arguments

- Arguments must be variables
  If declare: `void myFunc(float& inputNum);`
  - `myFunc(inVariable);` - GOOD syntax
  - `myFunc(25.4);` - BAD syntax

22

## Mixing parameters

- Can define a function that takes both values and references

```
void flipAndMult(int& num1, int& num2, int mult);
// flips num1 and num2 and multiplies each
// by mult
```

23

## More usage of &

```
int x = 5;
int& y=x; // y and x point to same address
y=10;
cout << x << endl; // output x value
cout << &x << endl; // output x address
```

24

## Procedural abstraction

- Function name stands in for set of statements
- Can use a function without knowing how it is written

```
int a=abs(-5);
float b=sqrt(2);
```

25

## Procedural abstraction, continued

What do we need to know?

- Function name
- Inputs
- Outputs

**Function declaration**

- Results of performing function

26

## Specifications

Preconditions:

- What is assumed to be true when function is called

Postconditions:

- What will be true after the function is called (presuming preconditions are met)
  - What values are returned
  - What call-by-reference parameters are changed
  - What other output is produced

27

## Example specification

- Include specs in comments of declaration

```
float sqrt(float inputNumber);
// Precondition: inputNumber is a
// positive float
// Postcondition: Function returns
// a float output such that output
// is non-negative and
// output*output=inputNumber
```

28

## What if a function calls itself?

```
int mysteryFunc3(int inNum) {
   if (inNum==0)
      return 2;
   else
      return inNum+mysteryFunc3(inNum-1);
}
```

29

## Recursion

When a function calls itself:

- Can be a simpler way to write a loop
- Can be used as a divide-and-conquer method

```
// Pseudo-code: outline of code design
findBiggestNum(num_list) {
   if (only one number in num_list)
      return number in num_list
   num1 = findBiggestNum(first half of num_list)
   num2 = findBiggestNum(second half of num_list)
   return max(num1,num2)
}
```

30

## Recursive function design

Must have:

- Base case(s) – to eventually stop recursion
- Simplified recursive calls – each new call must bring us closer to reaching base case(s)

31