CISC 1600/1610
Computer Science I

Functions: `void`, recursion, call-by-reference

Professor Daniel Leeds
dleeds@fordham.edu
JMH 328A

`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);`~~

Use of `return;`

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

Writing more compact programs using functions

From midterm:

```
 cout << numA;
  if (numA == 1)
    cout << " A\n";
  else
    cout << " As\n";

  cout << numB;
  if (numB == 1)
    cout << " B\n";
  else
    cout << " Bs\n";

  cout << numC;
          ⋮
```

Alternate

```
outputLet(numA,'A');

outputLet(numB,'B');

outputLet(numC,'C');

outputLet(numD,'D');
```

Recursion

When a function calls itself:

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

Alternate power function

```
int power(int num, int expon)
{
  if(expon>0)
    return num*power(num,expon-1);
  else
    return 1;
}
```

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)

What does this code do?

```
int funcC(int a);

int main() {
  int a;
  cout << "Enter a number: ";
  cin >> a;
  cout << funcC(a);
  return 0;
}

int funcC(int a) {
  if(a==0)
    return a;
  else
    return a+funcC(a-1);
}
```

Variable scope

Variables declared in a function

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

`int main()` is a function

What does this code do?

```
int newFunc(int a);

int main() {
  int a=5, b, c=5;
  b = newFunc(a);
  cout << a << " " << b << " "
       << c << endl;
  return 0;
}

int newFunc(int a) {
  int c=12;
  return a*5+c;
}
```

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

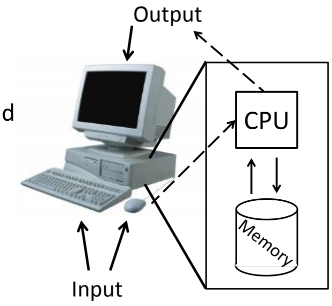
Computer system structure

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



Variables – locations in memory

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

	Address	Value	
repeatLoop	04902340	00000001	**true**
	04902348	00010110	
	04902356	11011101	
product (main)	04902364	00011000	**24**
	04902372	00100110	
	04902380	11011110	
	04902388	01000110	

Memory usage by functions

"Call-by-value":

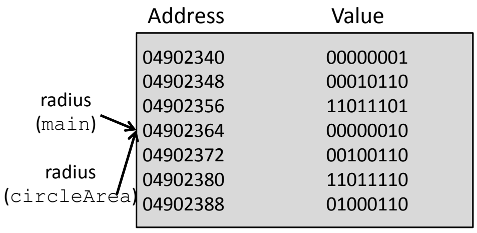
- provide function with the value held in a variable input
- assign value to new internal variable

	Address	Value
	04902340	00000001
	04902348	00010110
	04902356	11011101
radius (main)	04902364	00000010
	04902372	00100110
	04902380	11011110
radius (circleArea)	04902388	00000010

Memory usage by functions

"Call-by-reference":

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



Address	Value
04902340	00000001
04902348	00010110
04902356	11011101
04902364	00000010
04902372	00100110
04902380	11011110
04902388	01000110

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;
}
```

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

```
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?

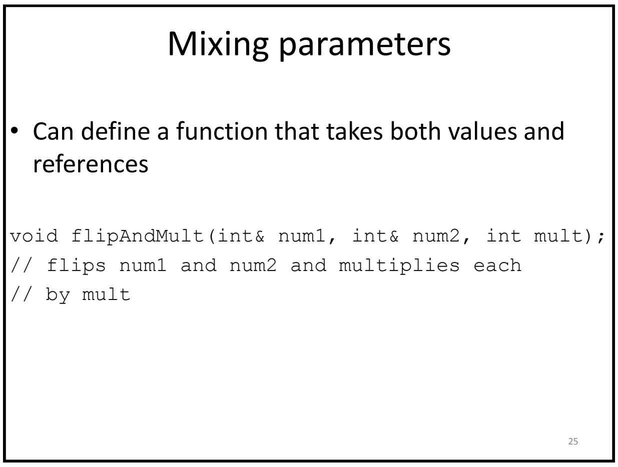
```
int mysteryFunc2(int inNum);

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?

Call-by-reference: Input arguments

- Arguments must be variables

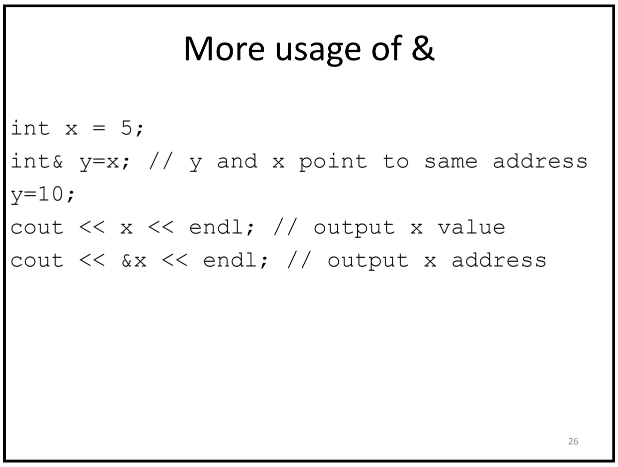
  If declare: `void myFunc(float& inputNum);`
  - `myFunc(inVariable);` - GOOD syntax
  - `myFunc(25.4);` - BAD syntax



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
```



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
```