

Introduction to Programming (CS 101)

Spring 2024



Lecture 9:

Functions (Part II)

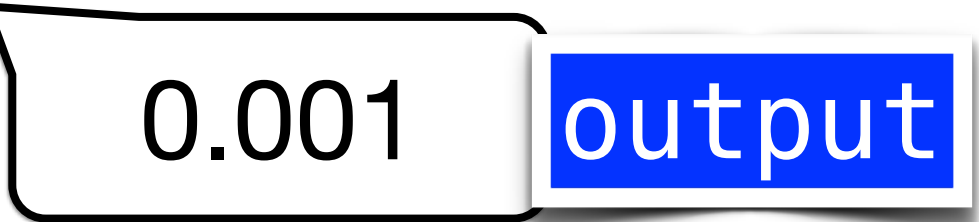
Instructor: Preethi Jyothi

Based on material developed by Prof. Abhiram Ranade and Prof. Manoj Prabhakaran

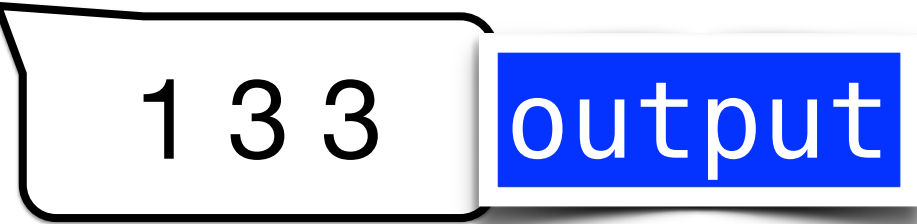
Recap (I)

What is the output of each of the following code snippets?

```
main_program {  
    cout << 0.001F << endl;  
}
```



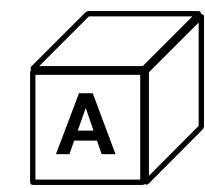
```
main_program {  
    int i = 1, j = 2, k = 3;  
    (i, j+=1) = (5, k);  
    cout << i << " " << j << " " << k;  
}
```



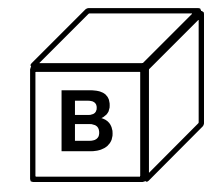
Recap (IIA)

What is the output of the following statement?

```
main_program {  
    float f1 = 2e7; //larger than  $2^{24}$   
    float f2 = 0.07;  
    double d = 1e-20; //smaller than  $2^{-53}$   
  
    cout << f1 + 1 - f1 << " " << 1 - f1 + f1 << endl;  
  
}
```



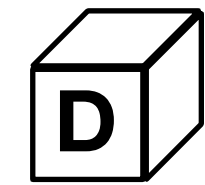
1 1



1 0



0 0



0 1

Recap (IIB)

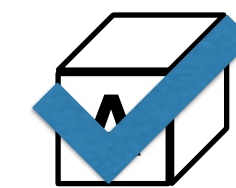
What is the output of the following statement?

```
main_program {  
    float f1 = 2e7; //larger than  $2^{24}$   
    float f2 = 0.07;  
    double d = 1e-20; //smaller than  $2^{-53}$ 
```

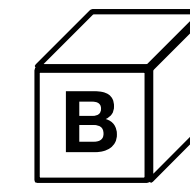
```
    cout << f1 + 1 - f1 << " " << 1 - f1 + f1 << endl;
```

```
    cout << 1 + f2 - 1 << " " << 1 - 1 + f2 << endl;
```

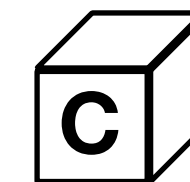
```
}
```



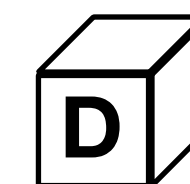
0.07000001 0.07



0.07 0.07



0.07 0.07000001



0.07000001 0.07000001

Recap (IIC)

What is the output of the following statement?

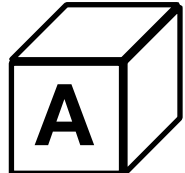
```
main_program {  
    float f1 = 2e7; //larger than  $2^{24}$   
    float f2 = 0.07;  
    double d = 1e-20; //smaller than  $2^{-53}$ 
```


```
    cout << f1 + 1 - f1 << " " << 1 - f1 + f1 << endl;
```

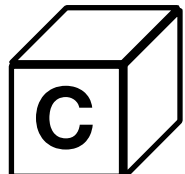
```
    cout << 1 + f2 - 1 << " " << 1 - 1 + f2 << endl;
```

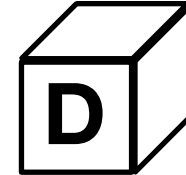
```
    cout << 1 + d - 1 << " " << 1 - 1 + d << endl;
```

```
}
```

 1e-20 1e-20

 0 1e-20

 1e-20 0

 0 0



Functions

CS 101, 2025

Recap of functions

- In a previous class, we wrote a program to compute `sin(x)` given `x`
 - Say you want to make repeated computations of `sin(x)` (and `cos(x)`).
 - E.g., Compute the sum of many sine and cosine waves at different frequencies, implement Euler's formula, etc.
 - Copy the relevant code (e.g. for `sin(x)`) wherever it's required?
 - Far from elegant, and more importantly error-prone!
- Functions are informally commands that compute values (`sqrt(x)`) or take actions (`forward(50)`)
 - Functions associate a **body** (sequence of statements) with a name and zero or more *function parameters*
- How do we define our own functions?

Syntax of functions

- Syntax: `return-type` `function-name`(`data-type1` `var-name1`, ..., `data-typen` `var-namen`)
 {
 body
 }

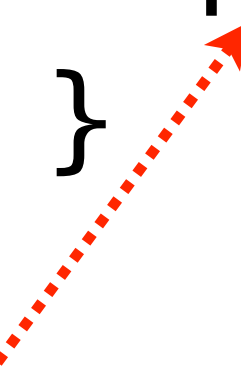
- Elements of a function:

- `return-type`: Function terminates by returning a value of type `return-type`
- `function-name`: User-defined name. Common to use mixed-case, and verbs. E.g., `setFlag`, `isEven`, etc.
- List of zero or more function parameters defined as `data-type1` `var-name1`, ..., `:`
Inputs to the function, together with their types.

- Example of a function:

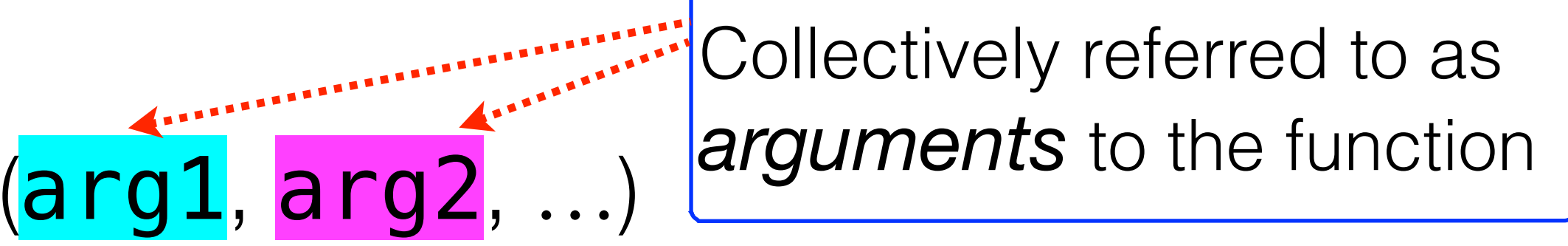
`return` terminates the function and returns a value of `return-type` to the function call

```
bool isEven(int n) {  
    return (n % 2 == 0 ? 1 : 0);  
}
```



Function calls (I)

- Function calls refer to an invocation of a newly defined function

- Syntax: `function-name (arg1, arg2, ...)`

Collectively referred to as *arguments* to the function

- Example:

```
bool isEven(int n) {  
    return (n % 2 == 0 ? 1 : 0);  
}
```

```
main_program {  
    int n;  
    cin >> n;  
    isEven(n) ? cout << "Even\n"; : cout << "Odd\n";  
}
```

Function calls (II)

- Default arguments
 - Value(s) provided in a function declaration that's automatically assigned by the compiler, if the function call does not provide value(s) for the argument(s)
 - This would allow for a function to be called without providing one or more arguments

- Example:

```
int addTen(int n, int t = 10) {  
    return (n + t);  
}
```

```
main_program {
```

```
    int n;
```

```
    cin >> n; //assume n = 3
```

output

13

```
    cout << addTen(n); //default value of t (i.e. 10) is used
```

```
    cout << addTen(n, 20); //20 overrides default value of t
```

```
}
```

Function calls (II)

- Default arguments
 - Value(s) provided in a function declaration that's automatically assigned by the compiler, if the function call does not provide value(s) for the argument(s)
 - This would allow for a function to be called without providing one or more arguments

- Example:

```
int addTen(int n, int t = 10) {  
    return (n + t);  
}
```

```
main_program {  
    int n;  
    cin >> n; //assume n = 3  
    cout << addTen(n); //default value of t (i.e. 10) is used  
    cout << addTen(n, 20); //20 overrides default value of t  
}
```

Function calls (III)

- Default arguments
 - Can be overwritten when the function call contains values for the default arguments
 - Arguments from calling function to called function are copied left to right
 - Default arguments should be assigned from right to left. E.g., `int addTen(int n = 1, int t)` is an error
- Note that function definitions should be placed *before* the main program
- Function *declarations*: Establish the name, return type, parameters of the function.
- Function *definitions*: Define the body of the function (allocates memory; shown later in the lecture)
- Function declarations appear before `main`, followed by function definitions after `main`

- Demo in class

main_program

- `main_program` you've been using so far is a function: `int main()`
- `main`'s return value is typically used as an error code by the shell
- Even if no explicit `return` statement, when the control reaches the end of the function, it implicitly returns the integer `0` (taken as no error by shells)
- Can explicitly return a non-zero value to indicate an error to the shell
- Note that the first line if you replace `main_program` with `int main()` would be `#include <iostream>` instead of `#include <simplecpp>`

void type

- When used as a return type: The `void` keyword specifies that the function doesn't return any value. Example of a declaration:
 - `void printCode(char c);` //definition should print character in c
- The argument list in a function call can be empty, if the function does not take any arguments
 - `int getInput();` //get integer input from user and return its value
- You cannot declare a variable of type `void`

Return types

- Every function definition must specify a return type, unless return type is `void`
- `void` return type indicates that the function should not return a value
- `void printMsg() { int i = 0; return i; }` will result in a compile-time error
- return value from a function will be type casted appropriately depending on the data type of the variable it's assigned to

What is the output of the following program?

```
float returnHalf() {  
    float f = 1.0/2; return f;  
}  
main_program {  
    int out = returnHalf();  
    cout << out;  
}
```

0

output



Detailed example with multiple functions

CS 101, 2025

Prime Factors Equivalence (PFE)

- Let us say two numbers are *prime-factors equivalent (PFE)* if they have exactly the same set of prime factors (ignoring multiplicities). Assume the two numbers are non-negative integers.
- Code template to check PFE:

```
int a, b;
```

```
cin >> a >> b;
```

```
bool a_covers_b; // a_covers_b if every prime factor of b divides a
```

```
bool b_covers_a; // similarly, b_covers_a
```

```
// TODO: code to evaluate a_covers_b
```

```
// TODO: code to evaluate b_covers_a
```

```
cout << ((a_covers_b && b_covers_a) ? "Equivalent!":"Not  
equivalent") << endl;
```

Prime Factors Equivalence (PFE)

```
int a, b;  
cin >> a >> b;  
  
// TODO: code to evaluate a_covers_b  
bool a_covers_b; // a_covers_b if every prime factor of b divides a  
  
a_covers_b = true;  
  
for (int d=2; b > 1; (b%d==0) ? b/=d : d++) {  
    if (b%d == 0 && a%d !=0) {  
        a_covers_b = false;  
        break;  
    }  
}
```

Prime Factors Equivalence (PFE)

```
int a, b;  
cin >> a >> b;
```

```
// TODO: code to evaluate a_covers_b
```

```
bool a_covers_b; // a_covers_b if every prime factor of b divides a
```

```
a_covers_b = true;
```

```
for (int d=2; b > 1; (b%d==0) ? b/=d : d++) {  
    if (b%d == 0 && a%d !=0) {  
        a_covers_b = false;  
        break;  
    }  
}
```

Alters b. Will need to work on a copy of b so as to not alter the original number in b.

```
// TODO: similarly, code to evaluate b_covers_a
```

```
// TODO: Repeat the above code snippet with a and b swapped!
```

Challenges with duplicating code

- Have to ensure a and b are carefully swapped
- If any bugs appear in one place, should remember to fix in both places
- If want to modify or augment the equivalence constraints, will need to update code in both places

PFE using a function

```
int a, b;  
cin >> a >> b;  b-->n, a--m
```

```
// TODO: code to evaluate a_covers_b
```

```
bool a_covers_b = covers(a, b); // define function covers(a,b)  
bool b_covers_a = covers(b, a);
```

```
cout << ((a_covers_b && b_covers_a) ? "Equivalent!":"Not equivalent") << endl;
```

Scope of m and n
are limited to the
function covers;
the original values
of a, b (passed to
covers) are not
modified

```
bool covers(int m, int n) {  
    for (int d=2; n > 1; (n%d==0) ? n/=d : d++) {  
        if (n%d == 0 && m%d !=0)  
            return false;  
    }  
    return true;  
}
```

PFE using two functions

```
main_program {  
  
    int a, b;  
    cin >> a >> b;  
    cout << PFE(a,b) ? "Equivalent!":"Not equivalent" << endl;  
  
}
```

```
bool covers(int m, int n) {  
    for (int d=2; n > 1; (n%d==0) ? n/=d : d++) {  
        if (n%d == 0 && m%d !=0)  
            return false;  
    }  
    return true;  
}
```

```
bool PFE(int a, int b) {  
    return covers(a,b) && covers(b,a);  
}
```



Next class: Functions and References
CS 101, 2025