C++
Programming Bootcamp 2

COSC2802

TOPIC 4 User Defined Functions

Functions Basics

Functions Basics

What are functions?

Similar concept to Java Methods

A block of code that can be called from elsewhere in the program repeatedly.

Why do we care?

Allows a lot of code reuse.

- Avoids copy/pasting
- Easy edits to reused code.
- Makes code easier to read

```cpp
#include <iostream>

void print() {
    std::cout << "PRINTING!" << std::endl;
}

int main() {
    print();
    print();
    print();
    print();
    return 0;
}
```

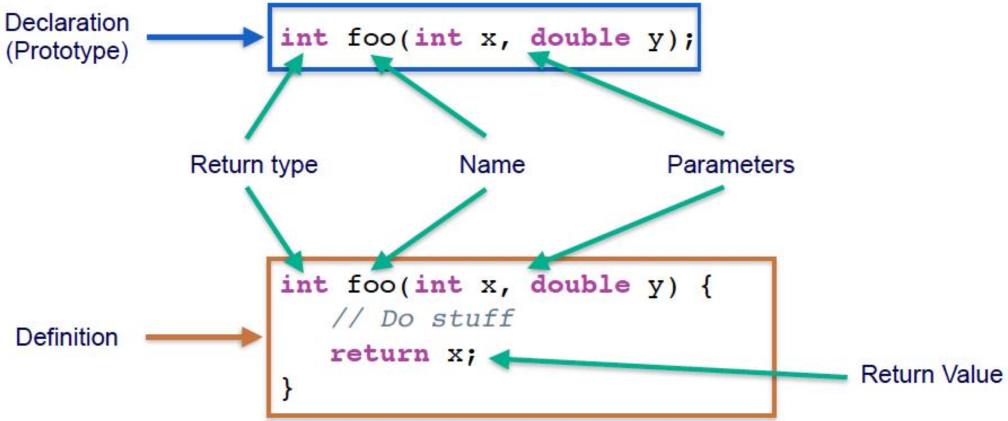
Function Structure

A function is made up of two parts: A declaration and a definition.

- A declaration just lets the compiler know that this function will exist at some point.
- The definition of a function is the actual implementation.

Both declarations and definitions require you to specify:

1. The type of data being returned (or void if no returned data)
2. The name of the function
3. And the parameters the function will accept



Parameters vs Arguments

- A parameter is the declaration of what variables a function will be passed.
- An argument is the actual value provided to the function when it is being called.

In this example:

i and *f* are parameters, while *a* and *2.0,* are arguments.

```
void func(int i, float f) {}

void main()
{
    int a = 1;
    func(a, 2.0);
}
```

Function Variations

Functions can have:

- Multiple parameters separated by commas, within parentheses

```
int foo(int x, float y);
```

- No parameters – parentheses still required

```
int main() {
```

Returning value from functions:

- A function may return one value using a return statement

```
return EXIT_SUCCESS;
```

- No return value – return type void

```
void PrintSummary(int id, int items, double price)
```

Default Parameters

- Default parameters (see zybooks)

```
void PrintDate(int currDay, int currMonth, int currYear, int printStyle = 0) {
    if (printStyle == 0) { // American
        std::cout << currMonth << "/" << currDay << "/" << currYear;
    }
    else if (printStyle == 1) { // European
        std::cout << currDay << "/" << currMonth << "/" << currYear;
    }
    else {
        std::cout << "(invalid style)";
    }
}

int main() {
    PrintDate(30, 7, 2012); // 7/30/2012
    return 0;
}
```

Function Overloading

You can have multiple versions of the same function but with different parameters combinations.

```
void PrintDate(int currDay, int currMonth, int currYear) {
    std::cout << currDay << "/" << currMonth << "/" << currYear <<
std::endl;
}

void PrintDate(int currDay, std::string currMonth, int currYear) {
    std::cout << currDay << " " << currMonth << " " << currYear <<
std::endl;
}

int main() {
    PrintDate(30, 7, 2012); // 30/7/2012
    PrintDate(30, "July", 2012); // 30 July 2012
    return 0;
}
```

Functions Usage

Distinction between **function declaration** (prototype) and **definition**

- Functions must be either **defined** or **declared (prototyped)** before they can be called
- Must be only **defined only** once
  - If the function has been declared (prototyped) before it is called, then the definition can be
    - **after** it is called
    - OR in a different file (more later)

Functions: Declarations and Definitions

Function definition

Defined before use

```
int foo(int x) { return x * x; }

int main() {
    int i;
    std::cin >> i;
    std::cout << foo(i) << std::endl;
    return EXIT_SUCCESS;
}
```

Functions: Declarations and Definitions

Function definition

Defined before use

```
int foo(int x) { return x * x; }

int main() {
    int i;
    std::cin >> i;
    std::cout << foo(i) << std::endl;
    return EXIT_SUCCESS;
}
```

Function Declaration (prototype)

Defined after use

Function definition

```
int foo(int x);

int main() {
    int i;
    std::cin >> i;
    std::cout << foo(i) << std::endl;
    return EXIT_SUCCESS;
}

int foo(int x) { return x * x; }
```

How functions work

(time for some major theory!)

Mechanics of function calling process

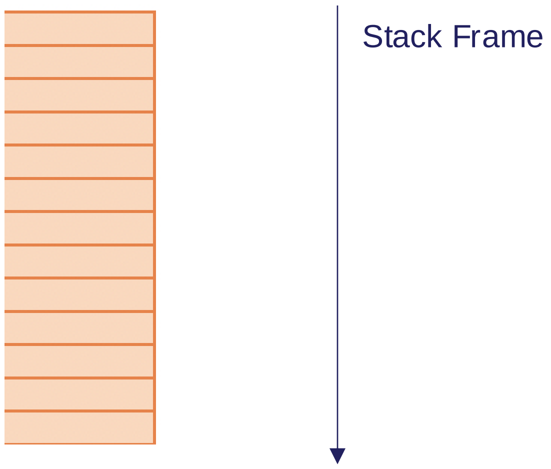
The following happens when a function is called **

1. calling program computes values for each ***argument*** - before the new function begins.
2. new space created for local function variables and ***parameter*** variables: the ***stack frame.***
3. value of each ***argument*** is ***copied*** into the corresponding ***parameter*** variable (type conversions are performed between the argument values and parameters if needed)
4. Statements in function body are executed until return statement or end of function
5. value of return expression, if any, is evaluated and returned (with type conversion if it doesn't precisely match the result type declared for the function)
6. ***Stack frame is discarded - all local variables disappear***.
7. calling program continues, with the **returned value** substituted in **place of the call**.'

** Taken from: Roberts pg 34 (Programming Abstractions in C++ - see Canvas)

How Functions work

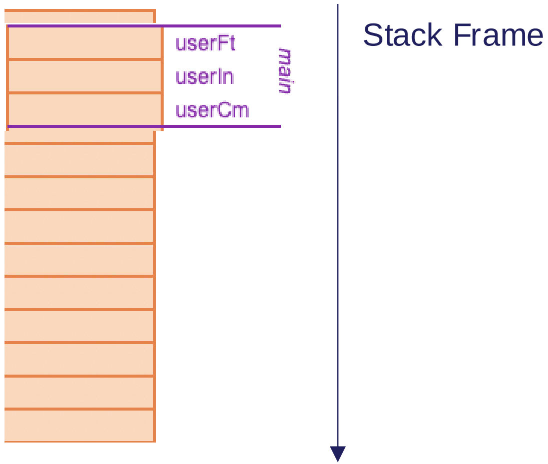
```
int FtInToIn(int inFeet, int inInches) {
   int totInches;
   ...
   return totInches;
}

double FtInToCm(int inFeet, int inInches) {
   int totIn;
   double totCm;
   ...
   totIn = FtInToIn(inFeet, inInches);
   ...
   return totCm;
}

int main() {
   int userFt;
   int userIn;
   int userCm;
   ...
   userCm = FtInToCm(userFt, userIn);
   ...
   return 0;
}
```



1. Each function call creates a new set of local variables – pushed onto stack
2. Each return causes those local variables to be discarded – popped off stack

How Functions work

```
int FtInToIn(int inFeet, int inInches) {
   int totInches;
   ...
   return totInches;
}

double FtInToCm(int inFeet, int inInches) {
   int totIn;
   double totCm;
   ...
   totIn = FtInToIn(inFeet, inInches);
   ...
   return totCm;
}

int main() {
   int userFt;
   int userIn;
   int userCm;
   ...
   userCm = FtInToCm(userFt, userIn);
   ...
   return 0;
}
```



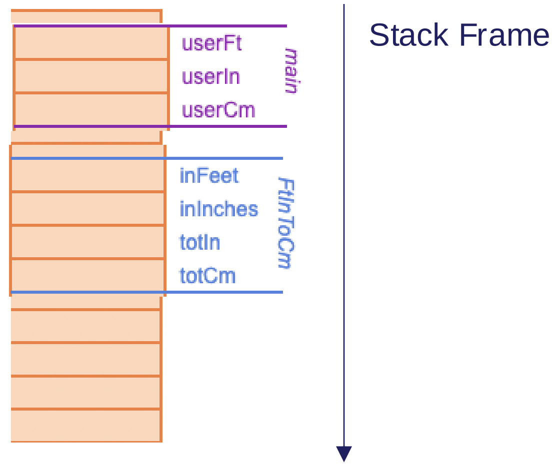
1. Each function call creates a new set of local variables – pushed onto stack

How Functions work

```
int FtInToIn(int inFeet, int inInches) {
   int totInches;
   ...
   return totInches;
}
```

```
double FtInToCm(int inFeet, int inInches) {
   int totIn;
   double totCm;
   ...
   totIn = FtInToIn(inFeet, inInches);
   ...
   return totCm;
}
```

```
int main() {
   int userFt;
   int userIn;
   int userCm;
   ...
   userCm = FtInToCm(userFt, userIn);
   ...
   return 0;
}
```



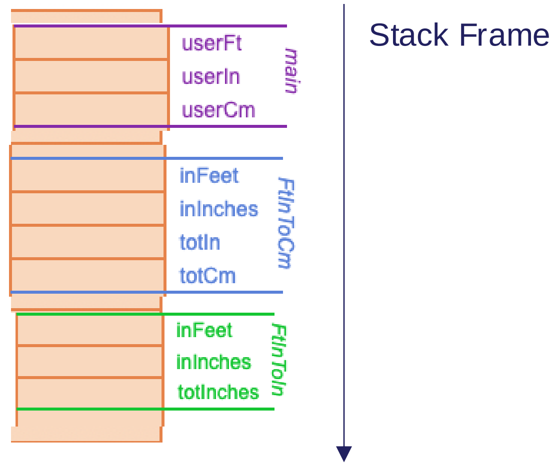
1. Each function call creates a new set of local variables – pushed onto stack

How Functions work

```
int FtInToIn(int inFeet, int inInches) {
   int totInches;
   ...
   return totInches;
}
```

```
double FtInToCm(int inFeet, int inInches) {
   int totIn;
   double totCm;
   ...
   totIn = FtInToIn(inFeet, inInches);
   ...
   return totCm;
}
```

```
int main() {
   int userFt;
   int userIn;
   int userCm;
   ...
   userCm = FtInToCm(userFt, userIn);
   ...
   return 0;
}
```



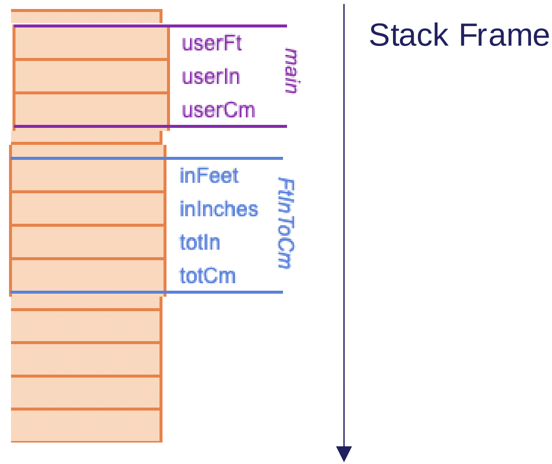
1. Each function call creates a new set of local variables – pushed onto stack

How Functions work

```
int FtInToIn(int inFeet, int inInches) {
   int totInches;
   ...
   return totInches;
}
```

```
double FtInToCm(int inFeet, int inInches) {
   int totIn;
   double totCm;
   ...
   totIn = FtInToIn(inFeet, inInches);
   ...
   return totCm;
}
```

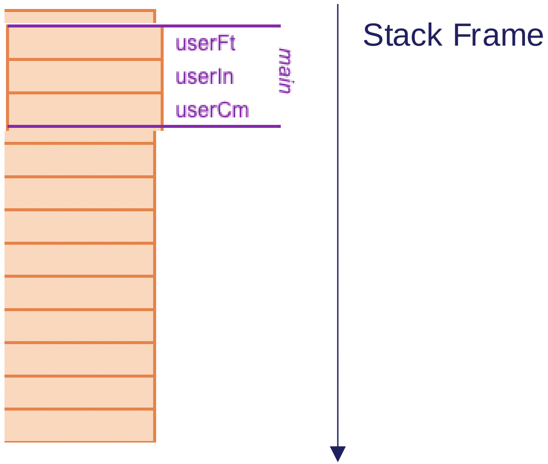
```
int main() {
   int userFt;
   int userIn;
   int userCm;
   ...
   userCm = FtInToCm(userFt, userIn);
   ...
   return 0;
}
```



1. Each function call creates a new set of local variables – pushed onto stack
2. Each return causes those local variables to be discarded – popped off stack

How Functions work

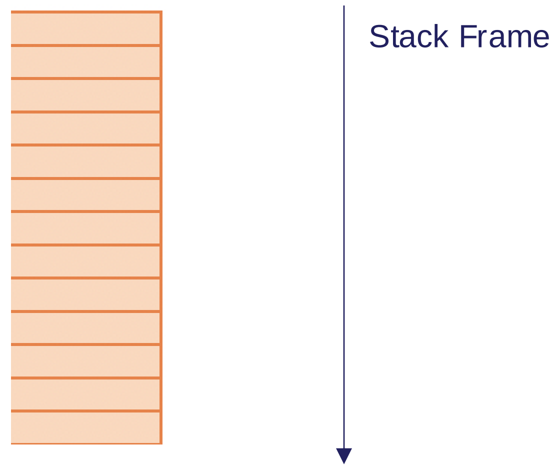
```
int FtInToIn(int inFeet, int inInches) {
   int totInches;
   ...
   return totInches;
}

double FtInToCm(int inFeet, int inInches) {
   int totIn;
   double totCm;
   ...
   totIn = FtInToIn(inFeet, inInches);
   ...
   return totCm;
}

int main() {
   int userFt;
   int userIn;
   int userCm;
   ...
   userCm = FtInToCm(userFt, userIn);
   ...
   return 0;
}
```



1. Each function call creates a new set of local variables – pushed onto stack
2. Each return causes those local variables to be discarded – popped off stack

How Functions work

```
int FtInToIn(int inFeet, int inInches) {
   int totInches;
   ...
   return totInches;
}

double FtInToCm(int inFeet, int inInches) {
   int totIn;
   double totCm;
   ...
   totIn = FtInToIn(inFeet, inInches);
   ...
   return totCm;
}

int main() {
   int userFt;
   int userIn;
   int userCm;
   ...
   userCm = FtInToCm(userFt, userIn);
   ...
   return 0;
}
```



1. Each function call creates a new set of local variables – pushed onto stack

Functions and Variable Scope

Scope: you can't see and access all the variables inside a program from anywhere else in the program!

Scope of **variables in functions**:

- limited to inside that function (variables currently on the stack frame)

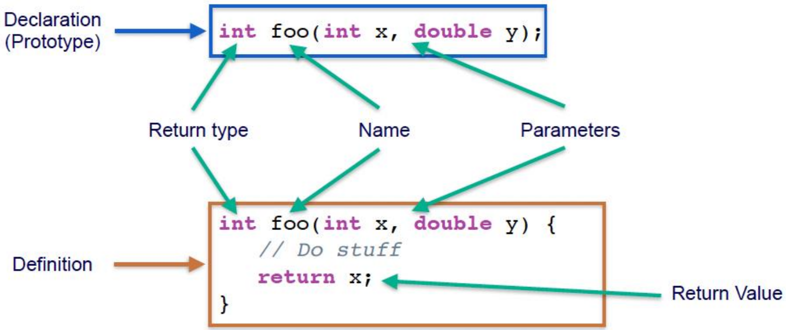
```
/* Converts a height in feet/inches to centimeters */
double HeightFtInToCm(int heightFt, int heightIn) {
   int totIn;
   double cmVal;

   totIn = (heightFt * IN_PER_FT) + heightIn; // Total inches
   cmVal = totIn * CM_PER_IN;                 // Conv inch to cm
   return cmVal;
}
```

Global variables are very bad practice and must be avoided!

Function Scope

Scope of **Functions**:

- extends from **declaration (prototype)** to the end of the file
- aim to have main() definition near the top of a file, with other functions defined below (or in separate files) … function declaration (prototype) preferred



Parameter Passing

i. **Pass by Value**

ii. **Pass by Reference (first look)**

iii. **Pass by Reference with Pointer (later n the course)**

iv. **…**

(i) pass-by-value

- A ***copy*** of the ***argument(s)*** is copied into the ***parameter(s)*** and used in the function.
  - Any modifications to the parameter inside the function will not affect the original argument outside the function.

What does the following print?

```
#include <iostream>

void foo(int x) {
    x *= x;
    std::cout << x << std::endl;
}

int main() {
    int i = 2;
    foo(i);
    std::cout << i << std::endl;

    return EXIT_SUCCESS;
}
```

(i) pass-by-value

Advantage:

- Because changes to the copy **do not** affect original value in caller - prevents accidental side effects that can affect correctness and reliability.

Disadvantage:

- if data item is large, can be expensive in execution time and memory space

(ii) pass-by-reference (first look)

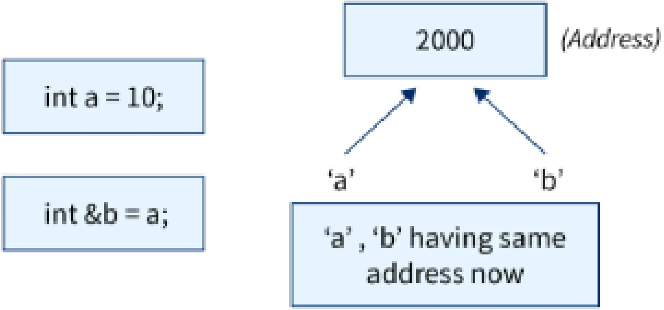
Reference Variable is an alias for an existing variable

- new name for same memory location as the existing variable
- i.e. **refers to the address** of another variable.

- Changes made to one affect the other – and vice versa

- Syntax for creating reference variable:

  *type***&** *ref_var = original_var;*   // *ref_va*r refers directly to location of *original_var*



(ii) pass-by-reference (first look)

Pass by Reference

- appending & to parameter's data type makes the parameter pass by reference
- Can make code more efficient/expressive as avoids copy overhead for large objects
- BUT: you can accidentally change variables which are needed for other parts of the program.

Also allows for some sneaky tricks regarding returning multiple values from a function!

(ii) pass-by-reference: Example

```cpp
#include <iostream>

void ConvHrMin(int totalMins, int& hours, int& minutes) {
    hours = totalMins / 60;
    minutes = totalMins % 60;
}

int main() {
    int totalMinutes = 156;
    int hours, minutes;

    ConvHrMin(totalMinutes, hours, minutes);

    std::cout << "Total Minutes: " << totalMinutes << ", is equal to: ";
    std::cout << hours << " hours and " << minutes << " minutes." << std::endl;

    return EXIT_SUCCESS;
}
```

constant pass-by-reference

To specify function doesn't change parameter use keyword ***const***

- get performance of pass-by-reference (good with large objects)
- Function treats parameter as constant (prevents assignment avoid side effects)

Example: zybooks 8.11.2

```
int foo(const int& a) {
    return a * a;
}
```

constant pass-by-reference

Example: see zybooks

```
void PrintVals(const vector<int>& vctrVals) {
   unsigned int i;  // Loop index

   // Print updated vector
   cout << endl << "New values: ";
   for (i = 0; i < vctrVals.size(); ++i) {
      cout << " " << vctrVals.at(i);
   }
   cout << endl;
}
```