ENGR101: Lecture 4

Functions in C++

ECS, VUW

March 2024

Reminders 1

- Everything is binary number
- Variables are stored in memory (naturally, as binary numbers)
- All variables should be declared (it specifies how many bits variable contains)
- Programs run by steps
- Any runnable C++ program should contain one **main()** - it is starting point of the program

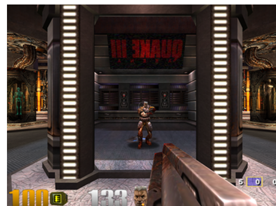
Introduction. Why use functions?

Many programs are BIG. As a rather naive metric we can count number of lines of code (LOC).

Quake 3 - 0.4 MLOC.
Windows XP - 45 MLOC
Mac OS Tiger - 86 MLOC

Source:

https://informationisbeautiful.net/visualizations/million-lines-of-code/

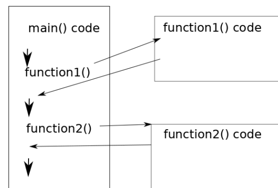


That would be impossible to write something like that as a single piece. Better approach is to separate the software into smaller parts. Smaller programs are called **function**.

What is the function in programming?

Function is separate piece of code, enclosed by braces with a name: **name**(){ function code here }.

- We write and test functions as separate pieces of code.
- Code execution is done line by line.
- When executed program reaches the line with function name - function is **called** and execution jumps inside the function.
- When last line of the function is reached - execution is transferred back. Function **returns**.



Function example

Listing 1: Function f1()

```
#include <iostream>
void f1(){
  std::cout<<"␣I␣am␣function"<<std::endl;
}

int main(){
   std::cout<<"␣I␣am␣main␣program"<<std::endl;
   f1();
   std::cout<<"␣I␣am␣main␣program"<<std::endl;
   return 0;
}
```

Have a look at this code...

Two similar blocks

Listing 2: Caption

```
#include <iostream>
using namespace std;
void f1(){
  cout<<'' I am function ''<<endl;
  }

int main(){
  f1();
  return 0;
}
```

- There are two blocks of code, similar in structure
- Structure is: type (**void** means no type), name/label (f1 and main), opening curly brace, some code and closing brace.
- Both blocks are **functions**

Execution order of programwith function

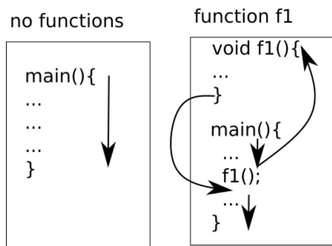


Figure: Execution flow

- If there are no functions - execution starts next line after **main()**{ and goes until paired }.
- There is a function:
  - Starts at same point
  - Reaches **f1();** line in **main()**. f1 is **called** - processor jumps to executing code between pair of brackets of **f1**.
  - Reaches closing brace in **f1** - jumps back to **main**.

Function definition and function call(use)

Listing 3: Caption

```
#include <iostream>
// definition
void f1(){
  std::cout<<"function"<<std::endl;
}

int main(){
   std::cout<<"main1"<<std::endl;
   f1();   //call
   std::cout<<"main2"<<std::endl;
   return 0;
}
```

- There is function **definition** - actual code of the function
- There is function **call** - use of the function
- C++ compiler is stupid here - it should know what function actually does before processing function call. Otherwise there is a compile error.
- It is limiting - **main()** should always be last in file text
- There is a way around it.

Function definition and function call(use)

Listing 4: Caption

```
#include <iostream>
//function declaration
void f1();

int main(){
    std::cout<<"main1"<<std::endl;
    f1();   //call
    std::cout<<"main2"<<std::endl;
    return 0;
}
// definition - actual code
void f1(){
   std::cout<<"function"<<std::endl;
}
```

We can put definition after call. But then we have to put **declaration** before call. Declaration in this case is **void f1();**. It is kind of promise to compiler: if there is call to **f1()** - there is defintion of it somehere, just look for it.

More usefull functions:

Function **f1()** as it is does not do much - it prints message on the screen.
It would be more usefull if function takes in some data, calculates something and gives result back.
To give result back - use **return** keyword.

Listing 5: give back

```
#include <iostream>
//function declaration
int f1();

int main(){
    int x = f1();   //call
    std::cout<<"x="<<x<<std::endl;
    return 0;
}

int f1(){
   return 2; //
}
```

Function arguments - getting data into the function

- Function **add** - takes two numbers and returns the sum
- We used names *a* and *b* for arguments at function definition
- When function is called, we used names *x* and *y*
- Is it right?

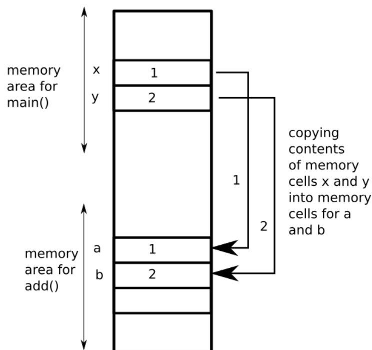
Listing 6: "Names"

```
#include <iostream>
using namespace std;
//definition
int add(int a, int b){
  int z;
  z = a + b;
  return z;
}

int main(){
  int x = 1;
  int y = 2;
  x = add(x,y); //call
  cout<<x<<endl;
}
```

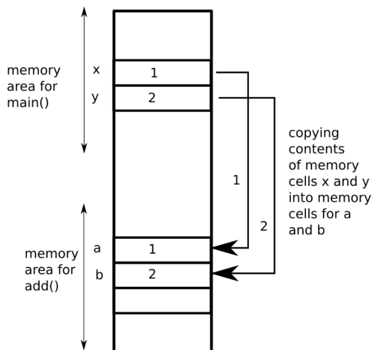
It is right - logic here is that function can be called many times with different arguments.

Memory for functions



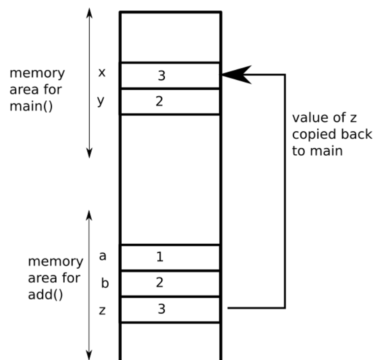
- When **main** starts - memory is reserved for variables **x** and **y**
- Values 1 and 2 are written into memory
- Code reaches line **x = add(x,y);**. **add()** is a function
- New memory area is used for all variables used in **add()**.
- There are two arguments (**a** and **b**) and **z**

Memory for functions



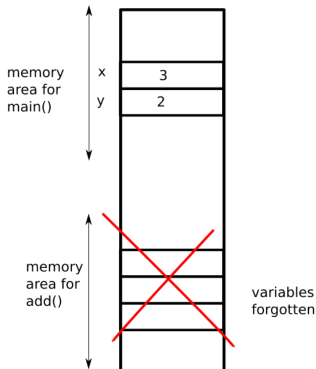
- Values of **x** and **y** are copied into memory area for **add()**
- In function call **add(x,y); x** is first in the list
- In function definition **int add(int a, int b) a** is first
- So value of **x** is copied into memory for **a**
- Same for variables which are second in function call and definition

function runs



- Code of **add()** runs
- Value of **z** is calculated and stored in memory
- **add()** code uses memory area for the function.
- **z** is created in this area
- Code reaches **return z** line
- Looks at line **x=add(x,y);**. Result of **add()** is copied into **x**

Finished with function




```
#include <iostream>
int add(int a, int b){
  int z = a+b;
  return z;
}

int main(){
  int x = 1;
  int y = 2;
  x = add(x,y); // x now is equal to x+y
```

we are here

- Memory area for **add()** is labeled as free to store other variables
- Result of such an arrangement is: whatever happens inside the function - stays inside the function.
- Very reasonable arrangement:
  - it limits number of variables you have to think about
  - it makes code speed optimization (cache)

Question?

Listing 7: Question

```
#include <iostream>
using namespace std;
void foo(int x){
  cout<<"Inside foo() x="<<x<<endl;
  x = x + 1;
  cout<<"Inside foo() x="<<x<<endl;
}

int main(){
  int x= 0;
  cout<<"In main(). x = "<<x<<endl;
  foo(x);
  cout<<"In main(). x = "<<x<<endl;
  return 0;
}
```

What will be printed?

1. In main(). x = 0
   Inside foo() x = 0
   Inside foo() x = 1
   In main(). x = 0
2. In main(). x = 0
   Inside foo() x = 0
   Inside foo() x = 1
   In main(). x = 1

Local variables

Varaiables declared inside the function (inside braces) - local variables.

Listing 8: Local vars

```
include <iostream>
using namespace std;
int foo(){
  int a  = 9;
   cout<<"a="<<a<<endl;
   return a;
}

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

You can (and should, really) declare variables inside the function. Such variables are called **local**. **Scope** of the variable is, as usual, from declaration until next closing bracket (highlighted). Outside of the **scope** variables do not exist.

Question

Listing 9: Caption

```
include <iostream>
using namespace std;
int foo(){
  int a  = 9;
  cout<<"a="<<a<<endl;
  return a;
}

int main(){
  foo();
  cout<<"a="<<a<<endl;
  return 0;
}
```

What happens if you run this program?

1. Prints:
   $a = 9;$
   $a = 9;$
2. Does not compile
3. Prints:
   $a = 9;$
   $a = 0$

Global variables - BAD ones

You can declare the variable outside of any pair of braces, like it is shown in listing below. Then this variable becomes **global** - it is visible and can be used anywhere in the program (after declaration).

Listing 10: Global a

```
include <iostream>
using namespace std;
int a;
int foo(){
 a = 9;
cout<<" a="<< a <<endl;
 return a;
}

int main(){
 foo();
cout<<" a="<< a <<a<<endl;
return 0;
}
```

What happens if you run this program?
It runs fine and **a** is defined everywhere in the program.
Now you have a problem: no matter which part of the program you edit, have to be aware about what is value of **a** now.

By value

What we described above is called passing arguments **by value**. Name makes it clear that **value**s are copied over form one variable to another.
It is nice and safe technique which allows programmer to think only about limited number of variables.
But there is one not so good thing about it - it requires copying of the values.
It can be not big deal if couple of bytes are moved over.
It becomes slow when big arrays are argument or result of the function.
Remember, we mentioned that memory is slow. OK, here copying huge arrays can slow your program down.

By reference

Look at these listings. Find the difference.

Listing 11: old

```
#include <iostream>
int add(int a, int b){
        int z =a+b;
        return z;
}

int main(){
  int x = 1;
  int y = 2;
  x = add(x,y);
  std::cout<<"x="<<x<<std::endl;
  return 0;
}
```

Listing 12: new

```
#include <iostream>
int add(int& a, int& b){
        int z =a+b;
        return z;
}

int main(){
  int x = 1;
  int y = 2;
  x = add(x,y);
  std::cout<<"x="<<x<<std::endl;
  return 0;
}
```

By reference

Listing 13: Caption

```
#include <iostream>
int add(int& a, int& b){
        int z =a+b;
        return z;
}

int main(){
  int x = 1;
  int y = 2;
  x = add(x,y);
  std::cout<<"x="<<x<<std::endl;
  return 0;
}
```

- Instead of contents of memory cell (i.e. value) under the hood address is passed into the function if argument specification contains **&**, as in **int& a**.
- So, even if names are different (**a** and **x** in this case, they **reference** same memory cells).
- Passing address (one number) as an argument into the function is much faster than copying many elements of the array

Argument by reference

One side-effect of passing argument by reference:

Listing 14: Caption

```
#include <iostream>
int add(int a, int& b){
        int z =a+b;
        b=b+1;
        return z;
}

int main(){
        int x = 1;
        int y = 2;
   std::cout<<"before-y="<<y<<std::endl;
        x = add(x,y);
   std::cout<<"after-y="<<y<<std::endl;
   return 0;
}
```

- In left listing both **y** and **b** reference same memory cells
- When **b** is modified inside **add()** function, **y** is modified too

Question

Listing 15: "Did it change?"

```
#include <iostream>
using namespace std;
int max(int& x, int y){
    int z;
   if (x > y) z = x;
   else z = y;
   x = x+ 1;
   return z;
}

int main(){
   int a =5;  int b = 6;  int c;
   cout<<"Before:  a="<<a<<" b="<<b<<"  b = "<<c<<endl;
   c = mmax(a,b);
   cout<<"After:  a="<<a<<" b="<<b<<"  b = "<<c<<endl;
}
```

What is an output?

- Before: a=5 b=6
  After: a=5 b=6
- Before: a=5 b=6
  After: a=6 b=6

Questions?