PIC 10A

Lecture 12: Functions II

functions calling other functions

It is quite common for a function to call another function.

Example:

```
double calculate_money(int checking_num, int savings_num)
{
    double checking_amount = checking(checking_num);
    double savings_amount = savings(savings_num);
    return checking_amount + savings_amount;
}
```

A function could even call itself. This is called recursion. More on that later.

From the example above we see that the functions `checking` and `savings` must be defined before the function `calculate_money`.

Functions calling other functions

Suppose function1 calls function2 which in turn calls function3.

We know that before function1 is defined function2 must be defined since the function definition of function1 mentions function2 and compiler has to know about the existence of function2. Similarly function3 has to be defined before function2.

General principle:

Functions have to be defined in reverse order.

Functions calling other functions

But what if a function1 calls function2 which calls function1?

By the general principle function2 has to be defined before function1 which means that function1 should be defined before function2...

This seems a to be a paradox. Luckily there is a way out.

Function prototypes

We can declare a function without defining it on the spot.

We promise the computer that eventually we will have a function by this name and having all these data types.

Function prototypes are placed above main.

Syntax:

type function_name(type par1, type par2,...);

Function prototypes

```
#include <iostream>
using namespace std;

int max ( int a, int b);

/* function definition can now be placed anywhere
   either before or after main.
   Order of functions does not matter. */

int max(int a, int b)
{
   if(a > b)
      return a;
   else
      return b;
}
```

Commenting functions

It is a good idea to comment all your functions. Good comment should indicate:

- What the function does
- What the input is
- What the output is

```
/*************************************************
** max: computes the maximum of two numbers (integers).
** Parameters:
** num1 -- The first number
** num2 -- The second number
** Returns the max of the two numbers.
*************************************************/
int max ( int num1, int num2 ) {
if( num1 > num2)
        return num1;
else
        return num2;
}
```

Scope and its benefits

Scope of a variable is the part of code where the variable lives.

For usual variables their scope is the function that they were declared in. Once the computer leaves the function the variable seizes to exist.

This is very useful because it means that the programmer does not have to keep track of variable names. If he used variable called x in one function he can re-use the name x in another function because the variables in different functions have nothing to do with each other.

Scope example

```
void foo ()
{
    int a=7;
    int b=0;
    string word="Hello";
    cout << a << " " << b << "\n"; // 7 0
    cout << c; // This line is an error!!

}

int main()
{
    int a=2;
    int b=3;
    int c=5;
    foo ();
    cout << a << " " << b << "\n"; // 2 3
    cout << word; // Illegal!!
    return 0;
}
```

Passing by value

When we pass variables to the function we create ***local copies*** of the variables.

We only pass the values of the variables in the function call to the local copies in the function. The local copies are independent of the variables in the function call.

The local copies may even have the same variable name as the variables in the function calls.

If a local copy of the variable changes its value, the original variable does not change its value!

Passing by value

```
int foo(int a, int b) //Note how parameters here are a and b
{
    cout << a << " " << b;
    a=10;
    b=20;
    return 5;
}

int main()
{
    int x=1,y=2,z=3;
    z = foo(x, y);  // Here we are calling the function with x and y
    cout << x << " " << y << " " << z;
    return 0;
}
```

Passing by value

```
int foo(int a, int b)
{
    cout << a << " " << b;
    a=10;
    b=20;
    return 5;
}

int main()
{
    int a=1,b=2,c=3;
    c = foo(a, b);
    cout << a << " " << b << " " << c;
    return 0;
}
```

Passing by value

```
int foo(int a, int b) //Note how parameters here are a and b
{
    cout << a << " " << b;
    a=10;
    b=20;
    return 5;
}

int main()
{
    int a=1,b=2,c=3;
    c = foo(b, a);
    cout << a << " " << b << " " << c;
    return 0;
}
```

Swap example: First try

```
void swap(double x, double y)
{
x=y;
y=x;
}

int main()
{
double x=2.;
double y=3.;
swap(x,y);
return 0;
}
```

This code is supposed to swap values of the variables x and y, but there are two things wrong here...

Swap: Second try

This does not work! We are copying y's value onto x, but then we copy that same value back to y! At the end x will just equal y.

```
void swap(double x, double y)
{
x=y;
y=x;
}
```

Ok. What if we use a temporary place holder!

```
void swap(double x, double y)
{
double temp=y;
y=x;
x=temp;
}
```

Swap cont.

This time our problem is that variables in functions have local scope. Changing them in functions is futile because their values will not have changed outside the function.

```
void swap(double x, double y)
{
double temp=y;
y=x;
x=temp;
cout << "value of x: " << x << " value of y: " << y << "\n";
}
int main()
{
double x=2.0;
double y=3.0;
cout << "value of x: " << x << " value of y: " << y << "\n";
swap(x,y);
cout << "value of x: " << x << " value of y: " << y << "\n";
return 0;
}
```

Passing by reference

We know that when we pass a variable to a function we are really just creating a local copy of that variable.

Sometimes this is incredibly inefficient. Our variable might in fact be a large class and making a copy of it would require a lot of effort.

Sometimes we do want the function to change our variables.

The solution is to pass our variable by reference.

Passing by reference

When we pass by reference we are not creating a local copy. You can think that we are actually passing the variable itself!

In reality what we pass is the memory address of our variable.

Syntax for function where we pass by reference

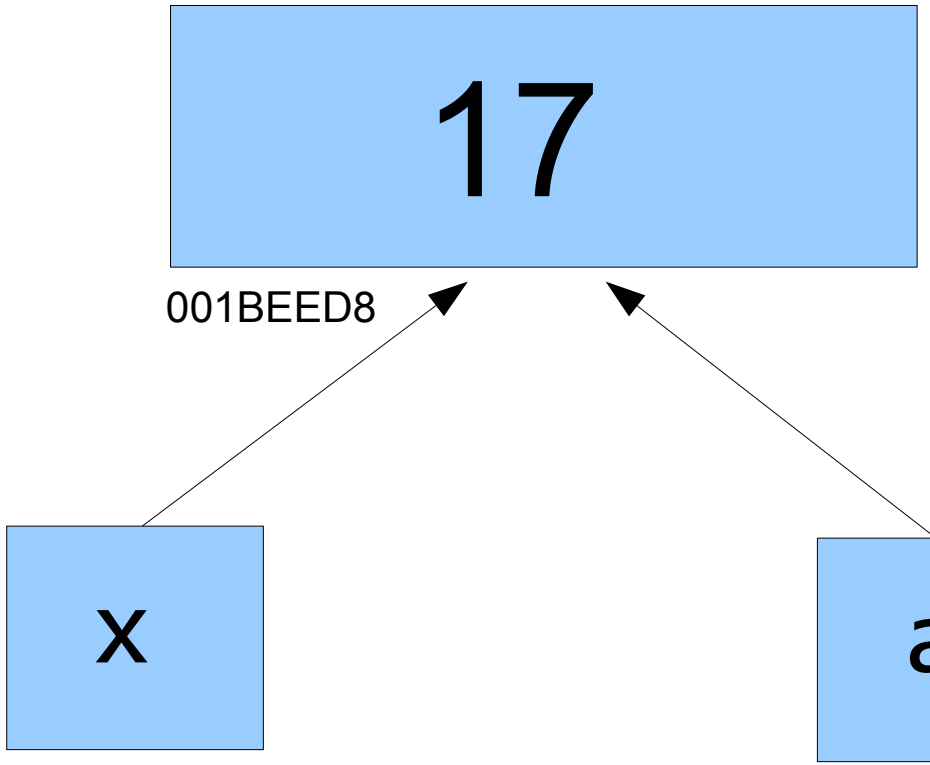
```
int foo(int &a)
{
// code
}
```

We call the function normally:

```
foo(x);
```

Passing by reference

The variable a in the function foo and the variable x in main are the same variable because they both share the same memory location.



Think of the memory location like a locker and both x and a use the same locker. Either one can change the contents of the locker.

Third time's a charm...

```
void swap(double &x, double &y)
{
double temp=y;
y=x;
x=temp;
cout << "value of x: " << x << " value of y: " << y << "\n";
}

int main()
{
double x=2.0;
double y=3.0;
cout << "value of x: " << x << " value of y: " << y << "\n";
swap(x,y);
cout << "value of x: " << x << " value of y: " << y << "\n";
return 0;
}
```

Keep track of variable values

```
int my_func(int a, int b, int& c)
{
    int d=a+b+c;
    a *=2;
    --b;
    c=2*(a+b);
    cout << a << " " << b-- << " " << c << " " << d << "\n";
    d--;
    return d;
}

int main()
{
    int a=1,b=2,c=3,d=4;
    a = my_func(a,b,c);
    cout << a << " " << b << " " << c << " " << d << "\n";
    return 0;
}
```

Solution

```
int my_func(int a, int b, int& c)
{
    int d=a+b+c; // local d is 1 + 2 + 3 = 6
    a *=2; // local a is 2
    --b; // local b is 1
    c=2*(a+b); // c is changed to 2*(2+1) =6 in my_func and also in main
    cout << a << " " << b-- << " " << c << " " << d << "\n"; // 2 1 6 6
    // local b is changed to 0
    d--; // local d is 5
    return d;
}

int main()
{
    int a=1,b=2,c=3,d=4;
    a = my_func(a,b,c); // a gets assigned 5
    cout << a << " " << b << " " << c << " " << d << "\n"; // 5 2 6 4
    return 0;
}
```

Example

```
void my_func(int &x)
{
   x*=2;
}

int main()
{
   int a=1;
   my_func(a);
   cout << a;
   return 0;
}
```

Old exam question: Part a)

```
int foo(int &a, int b, int &c)
{
    cout << a << " " << b << " " << c << "\n";
    a += 1+c;
    b *= 2;
    cout << a << " " << b << " " << c << "\n";
    return a;
}

int main ( )
{
    int a=20, b=30, c=40, d;
    cout << a << " " << b << " " << c << "\n";
    foo(a,b,c);
    cout << a << " " << b << " " << c << "\n";
    return 0;
}
```

Solution

```
20 30 40
20 30 40
61 60 40
61 30 40
```

Old exam question: Part b)

```
int foo(int &a, int b, int &c)
{
    cout << a << " " << b << " " << c << "\n";
    a += 1+c;
    b *= 2;
    cout << a << " " << b << " " << c << "\n";
    return a;
}

int main ( )
{
    int a=20, b=30, c=40;
    cout << a << " " << b << " " << c << "\n";
    foo(c,b,a);
    cout << a << " " << b << " " << c << " " << "\n";
    return 0;
}
```

Solution

```
20 30 40
40 30 20
61 60 20
20 30 61
```

Old exam question: Part c) (AKA Did I blow your mind yet?)

```
int foo(int &a, int b, int &c)
{
    cout << a << " " << b << " " << c << "\n";
    a += 1+c;
    b *= 2;
    cout << a << " " << b << " " << c << "\n";
    return a;
}

int main ( )
{
    int a=20, b=30, c=40;
    cout << a << " " << b << " " << c << "\n";
    foo(b,b,b);
    cout << a << " " << b << " " << c << "\n";
    return 0;
}
```

Solution

```
20 30 40
30 30 30
61 60 61
20 61 40
```