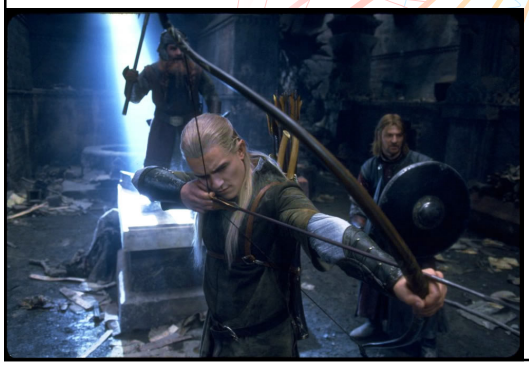
Lecture 12: Passing to Functions



PIC 10A

Todd Wittman

A Comment on Comments

- When commenting your homework,

  QUANTITY≠QUALITY !!!

- Some students are commenting every line of code, but the comments are mostly meaningless.

```
int x = 1;   //Sets integer x=1.
while ( x < 20 ) {   //This is a while loop.
      cout << x << " ";   //Outputs x and " ".
      x = x + 2;   //Adds 2 to x.
}  //Ends the while loop.
```

- Comments are meant to help the reader understand the code, not make it harder to understand!

A Comment on Comments

- Would have been much better to just comment on the block of code, rather than commenting every line.

```
// Print out the odd numbers 1 through 19.
int x = 1;
while ( x < 20 ) {
        cout << x << " ";
        x = x + 2;
}
```

- You were told it's OK to over-comment your HW, but this assumes you write worthwhile comments.

Sec 4.4: Return Values

- Your function should always return something, unless it's a void.
- For example, a square root is only defined for non-negative numbers (in the real numbers).
- One root of the quadratic formula is: $x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$

```
double compute1Root (double a, double b, double c)  {
        if ( b*b - 4*a*c >= 0 )
                return (-b+sqrt(b*b-4*a*c))/(2*a);
}
```

- But this returns nothing for if b*b-4*a*c < 0. Not good!

```
double compute1Root (double a, double b, double c) {
        if ( b*b - 4*a*c >= 0 )
                return (-b+sqrt(b*b-4*a*c))/(2*a);
        return 0;
}
```

Default return value. Could be anything you want.

Function Declarations

- Last lecture we mentioned that all functions have to be written before the main routine, because C++ reads the code top to bottom.
- When you have multiple functions and functions calling other functions, they should be listed in reverse order in which they're called.
- This can get tricky for complex programs.
- One way around this is to declare your functions right after the namespace line.

```
void my_function (double x, int y, string s);
```

- The declaration looks just like the prototype starting the function, except with a semi-colon.

Two Ways to Make Functions Work

- Order the functions in reverse order in which they're called.

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

int add2Numbers (int a, int b)  {
    return a+b;
}

int main ( )  {
    cout << "2+3=";
    cout << add2Numbers(2,3);
    return 0;
}
```

OR

- Declare the functions at the top. Then the function order doesn't matter.

```
#include <iostream>
using namespace std;
int add2Numbers (int a, int b);

int main ( )  {
    cout << "2+3=";
    cout << add2Numbers(2,3);
    return 0;
}

int add2Numbers (int a, int b)  {
    return a+b;
}
```

Declaration
Sometimes we make the declarations in a separate header file.

Prototype
Looks just like declaration.

Sec 4.6: Side Effects

- It's standard convention for the function not to have any visible side effect. Things like cout statements are generally left to main( ).

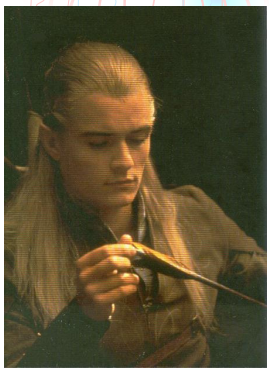
```
int doubleNumber (int x) {
        x = x*2;
        cout << x;      //Visible side effect.
        return x;
}
```

- This makes the function more re-useable.
- But of course there are exceptions:

    DrawX (p), DrawO (p), DrawMap( )

Sec 4.5: Value Parameters

- We pass the value of the parameter, not the actual variable.
- So the function creates a local copy of the variable just for that function.
- Changing the value of the passed parameter does not affect the value of the variable stored in another function.



What is the output of this program?

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

int my_fun (int a, int b, int c) {
    a = 10*a;
    b = 42;
    cout << "In my_fun, a="<<a<<" b="<<b<<" c="<<c<<"\n";
    return a+b+c;
}
```

OUTPUTS

In my_fun, a=20 b=42 c=4
In main, a=2 b=3 c=66

Value Parameters

- Generally, a good programming style is to treat value parameters in a function like constants. Don't even try to change them because the changes won't show up outside the function.
- Some programmers use different variable names in the function and the caller:

  In function: double my_fun (int a, int b, int c) {

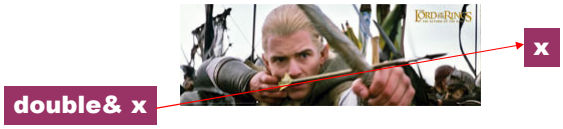
  In main: double w = my_fun (x, y, z);
- If the variable is large (like a database or an image), making copies of it every time the function is called will eat up a lot of memory and copying time.
- Plus, sometimes we want to be able to change the value of the parameter.
- The solution is to pass the reference to the parameter.

Sec 4.8: Reference Parameters

- A reference parameter passes the address of the variable.
- We denote the address with & right after the parameter's data type.

```
void my_function (double& x)
```

- Any changes to x within my_function will now be reflected in the function that called my_function.



Reference Parameters

- We use reference parameters if we want to change the value of the parameter.
- For example, add one month's interest to a bank account. The amount in the bank should be updated.

```
void addMonth ( double& amount, double rate ) {
    amount = amount * (1 + rate);
    return;
}
```

```
double my_amount = 100.0;
addMonth ( my_amount, 0.05 );
cout << "You now have $" << my_amount;
```

- In this example, we could've made the new amount the return value. But it makes more sense to update the variable amount.
- What would we happen if we said: addMonth(100.0, 0.05) ?

Reference Parameters

- We should use reference paramters if want to return more than one value.
- Recall our quadratic formula example only returned one root.
- What if we want to return both real roots?
- One solution is to make the roots reference parameters.

```
void compute2Roots (double a, double b, double c,
        double& root1, double& root2) {
            if ( b*b - 4*a*c >= 0 ) {
                root1 = (-b + sqrt(b*b-4*a*c))/(2*a);
                root2 = (-b - sqrt(b*b-4*a*c))/(2*a);
            }
            return;
}
```

```
double x1, x2;
compute2Roots (2,10,4,x1,x2);
```

Note x1, x2 are uninitialized.

What is the output of this program?

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

int my_fun (int a, int& b, int c) {
    a = 10*a;
    b = 42;
    cout << "In my_fun, a="<<a<<" b="<<b<<" c="<<c<<"\n";
    return a+b+c;
}
```

OUTPUTS
In my_fun, a=20 b=42 c=4
In main, a=2 b=42 c=66

Just added one &

Reference Parameters

- Note the caller doesn't have to use the same variable name as in the function. Reference parameters can still change the value of the variable used.

```
int main ( ) {
   int x=2;   int y=3;
   int z = my_fun (x, y);
   cout<<"x="<<x<<" y="<<y<<" z="<<z<<"\n";
   return 0;
}

int my_fun (int& a, int b) {
   a=10*a;
   b = 42;
   cout<<"a="<<a<<" b="<<b<<"\n";
   return b / a;
}
```

OUTPUTS
a=20 b=42
x= 20 y=3 z=2

Functions Are Your Friend

- **Makes the code easier to read and understand.**
- **Makes it easier when you're programming to compartmentalize your tasks.**
- **Makes it easier when you're debugging to track down and fix your errors.**
- **Avoids repeating code.**
- **Don't be afraid of functions!**

Functions Are Not Your Friend

- **The function passing examples make nice test problems. Track the variables carefully.**

```
int fun (int a, int& b) {
  a = 3;
  b = 2*a+b;
  cout <<"In fun, a="<<a<<" b="<<b<<".\n";
  return a-b;
}

int main ( ) {
  int a=1;  int b=3;
  a = fun (a,b);
  cout << "In main, a="<<a<<" b="<<b<<".\n";
  return 0;
}
```

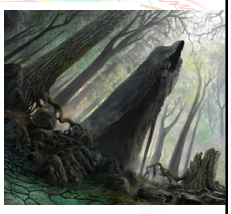
OUTPUT
In fun, a=3 b=9
In main, a=-6 b=9

- **Be afraid of functions!**

A Harder Function Passing Example

```
int divThis (int a, int& b, int c)  {
   int x = 5;
   a = a+2*b*5;
   b = b +x - 2;
   c = a*c-b;
   cout << "In divThis, a="<<a<<" b="<<b<<"
   c="<<c<<".\n";
   return a/b;
}

int main ( ) {
   int a=2;  int x=3;
   a = divThis(x,x,a);
   cout << "In main, a="<<a<<" x="<<x<<".\n";
   return 0;
}
```



OUTPUT
In fun, a=33 b=6 c=60
In main, a=5 x=6

Scope

- The scope of a variable is the part of the program in which the variable is defined.
- In the last example…
- … the variable a was a value parameter, so there were two copies of a. One had scope in main, the other was only within the function divThis.
- … the variable c is only defined in divThis, so it's scope is just the function divThis.
- … the variable x was passed by reference, so it's scope was both main and divThis even though it's name changed locally from x to b. Any change to b in divThis would change x in main.

Global Variables

- What if we want a variable to have global scope, so that every function can use it?
- One solution is to pass the variable as a reference parameter to every function.
- But this is kind of annoying.
- Another solution is to declare it as a global variable at the top of your program, outside a function.

```
using namespace std;
int x;
int main ( ) { ….
```

- Every function can use x. Any change to x anywhere will be recorded.

```
#include <iostream>
#include <string>
using namespace std;
int x;
string s;

int fun ( int a, int& b ) {
    s = "bal";
    b += 3;
    x = a+b;
    return 2*a;
}

int main ( ) {
    int a=1;    int b=2;
    b = fun (b, a);
    s = s + "rog";
    cout << "a="<<a<<" b="<<b<<" x="<<x<<" s="<<s;
    return 0;
}
```

OUTPUT
a=4 b=4 x=6 s=balrog