CALL-BY-REFERENCE

Today

- Today we will cover one of the trickiest things about functions
  - Call-by-reference
- We will also discuss two new control structures:
  - The do/while loop
  - The conditional operator

Reference parameters

- Last class we talked about how functions can pass parameters
  - How the value of those parameters might change inside the function
  - But in the *calling function*, the value of the parameters does not change
- We also talked about *scope*
  - How the variables we have defined have scope local to the function they are called in.
  - How these *local* variables "go away" when the function exits

- *call-by-value* example:

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

int add( int a, int b ) {
  int ret;
  ret = a + b;
  return( ret );
} // end of add()

int main() {
  int p = 7, q = 5, sum;
  sum = add( p, q );
  cout << "sum=" << sum << endl;
} // end of main()
```

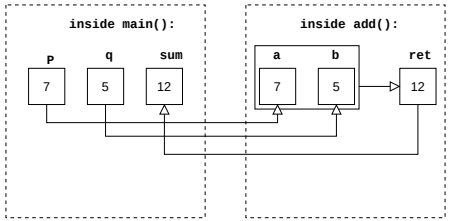
```
#include <iostream>
using namespace std;

int add( int a, int b ) {
  int ret;
  ret = a + b;
  return( ret );
} // end of add()

int main() {
  int p=7, q=5, sum;
  sum = add( p, q );
  cout << "sum=" << sum << endl;
} // end of main()
```

value parameters:
the initial values of a and b are set based on the values of the arguments used to call the function

return value:
the value of "ret" is returned by the function to the caller

function arguments:
the values of p and q are used to initialize the values of the parameters (a,b) inside the function

function return:
the value of "sum" is assigned to the function's return value



- In C++, there is a feature of functions called *reference parameters*.
- This lets you pass what is called the "address" of a variable to a function.
- This means that it is the variable itself rather than a copy that gets passed to the function.
- As a result, when the function exits, if the value of the variable has changed inside the function, then the new value can be retained outside the function.

- The classic example of using reference parameters is a function called `swap()`

```
void swap( int &a, int &b )
{
  int tmp;
  tmp = a;
  a = b;
  b = tmp;
  return;
}
```


- The operation of this slightly different function is much different:

```
void noSwap( int c, int d )
{
  int tmp;
  tmp = c;
  c = d;
  d = tmp;
  return;
}
```

- The best way to get to grips with reference parameters is to write some code that uses them.

- Another example

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

void add( int a, int b, int &sum ) {
  sum = a + b;
} // end of add()

int main() {
  int p = 7, q = 5, sum;
  add( p, q, sum );
  cout << "sum=" << sum << endl;
} // end of main()
```

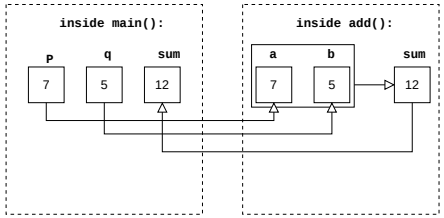


```
#include <iostream>
using namespace std;

void add( int a, int b, int &sum ) {
  sum = a + b;
} // end of add()

int main() {
  int p=7, q=5, sum;
  add( p, q, sum );
  cout << "sum=" << sum << endl;
} // end of main()
```

value parameters:
the initial values of a and b are set based on the values of the arguments used to call the function

reference parameter:
the value of "sum" is changed both inside the function and within the scope of the caller

function arguments:
the values of p and q are used to initialize the values of the parameters (a,b) inside the function; the value of sum is changed inside the function call and, because it is a reference parameter, the new value is retained when the function exits and assigned to the value of the argument in main()



The `do/while` loop

- If we want to read six numbers from the `ifstream` `myfile` we might use a `for` loop:

```
for(counter = 0; counter < 6; counter++)
  {
    myfile >> numbers[counter];
  }
```

- This is fine so long as we know that there are at least six numbers in the file.
- Sometimes we are not so lucky.
- We want to read in up to six numbers, but there may not be as many as that.

- In such a case (which is not uncommon) we might want to do the following:

```
do {
    myfile >> numbers[counter];
    counter++;
  }
while(counter < 6 && myfile);
```

- The second `myfile` is `true` so long as the last read from the file returned something.
- Thus when we get to the end of the file, `myfile` is false, and the loop ends.

- This is an example of a `do/while` loop.
- This is subtly different from a `while` loop.
- The general form of the loop is:

```
do {

    <some instructions>

}
while(<a condition that is true or false>);
```

- Note the semicolon at the end of the line with the `while` on it.

- The big difference between `do/while` and `while` is how many times the *body* of the loop gets repeated.
  - The body of the loop is the bit between the { and the }
- In a `while` loop, if the condition is `false`, the body is *never* executed.
- In a `do/while` loop, even if if the condition is `false`, the body is executed *at least once*.
- This difference helps us to decide which control structure is best to use.

- Another way to achieve the same thing is to do:

```
while(counter < 6 && !myfile.eof()) {
  myfile >> numbers[counter];
  counter++;
}
```

- The function `myfile.eof()` returns true if we are at the end of the file.
- Thus, once again, when we get to the end of the file, the loop terminates.

The conditional operator

- C++ contains a compact version of `if else`, which can sometimes be useful.
- `<condition> ? <if true> : <if false>`
- If the condition is true, the bit between the `?` and the `:` gets executed.
- If the condition is false, the bit between the `:` and the `;` gets executed.

- Thus

```
if (a >= b)
{
  y = a;
}
else
{
  y = b;
}
```

can be written as:

```
y = a <= b ?  a :  b;
```


Summary

- This lecture has described *call-by-reference*, when we use *reference* parameters.
- The behavior of call-by-reference is in contrast to what we see when we use the (normal) *call-by-value*.
- This lecture also looked at do/while loops and the conditional operator.