cis1.5 spring2009 lecture III.3

today we are going to talk about...

- functions and parameters
  - "call by value"
  - "call by reference"
- variable **scope**
- library functions for formatted output

functions and parameters

- last class we talked about how functions can pass *parameters*
- inside the function, the parameters are treated like variables
- when the function is called, the values of the arguments are used to initialize the values of the parameters inside the function
- when the function exits, there are two ways to handle the parameter values:
  - with **call by value** or **value** parameters, any changes to the parameter values made inside the function are *not* retained outside the function;
    $\Rightarrow$ *the values of the arguments do NOT change*
  - with **call by reference** or **reference** parameters, changes to the parameter values made inside the function *are* retained outside the function;
    $\Rightarrow$ *the values of the arguments DO change*

value parameters example

- **call by value** parameters: 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()
```


- **call by value** parameters: example with annotations

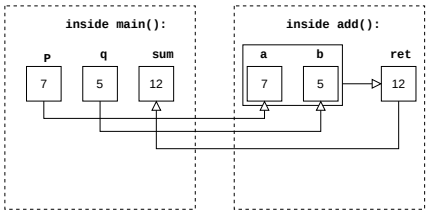
```
#include <iostream>
using namespace std;

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



- **call by value** parameters: same example as above, but using shorthand

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

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

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

- the arguments in `main()` are constants (`7, 5`), which are used to initialize the variables (`a` and `b`) inside the function `add()`

variable scope

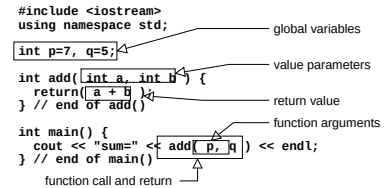
- variables are defined within either a *global* or a *local* scope
- *local* variables are defined inside a function and these "go away" when the function exits
- *global* variables are defined outside of any function, and these do not go away (as long as the program is running)
- in the first example:
  - `a` and `b` are local variables declared inside `add()`;
    their *scope* is the function `add()`;
    when `add()` exits, `a` and `b` no longer exist
  - `p` and `q` are local variables declared inside `main()`;
    their *scope* is the function `main()`;
    they also go away when `main()` exits, which is the same thing as when the program exits, because `main()` is the special function that controls the program

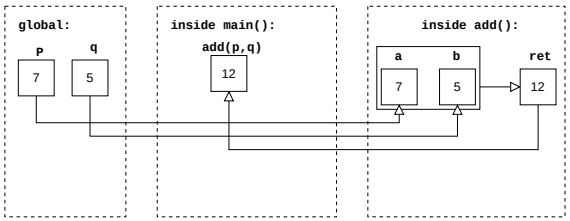
global variables example

- example similar to those above, except using global variables

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

int p = 7, q = 5; // declare global variables

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

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


- example with annotations:



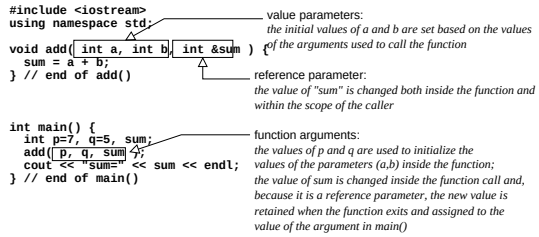


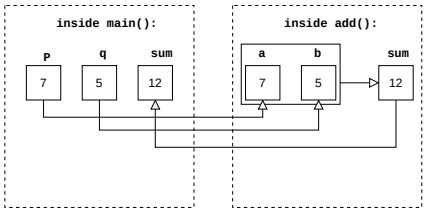
reference parameters example

- this example does the same thing as those above, except that it uses a reference parameter
- notice the *ampersand* (&) preceding sum in the function header—this indicates that it is a *reference* parameter (the & is the only indicator)

```
#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()
```


- example with annotations:





classic example, "swap", which uses reference parameters

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

void swap( int &a, int &b ) {
  int tmp;
  tmp = a;
  a = b;
  b = tmp;
  return;
} // end of swap()

int main() {
  int p = 7, q = 5;
  cout << "before: p=" << p << " q=" << q << endl;
  swap( p, q );
  cout << " after: p=" << p << " q=" << q << endl;
} // end of main()
```


compare with "noswap", which uses value parameters

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

void noswap( int a, int b ) {
  int tmp;
  tmp = a;
  a = b;
  b = tmp;
  return;
} // end of noswap()

int main() {
  int p = 7, q = 5;
  cout << "before: p=" << p << " q=" << q << endl;
  noswap( p, q );
  cout << " after: p=" << p << " q=" << q << endl;
} // end of main()
```

formatted output

- part of `iostream` library
- `cout.setf()` defines the type of output field
- `cout.precision()` sets the decimal precision (for real numbers)
- `cout.width()` sets the width of the output field
- example:

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

int main() {
  cout << "here's a table with lined-up columns:\n";
  cout.width( 10 );
  cout << "monday";
  cout.width( 10 );
  cout << "tuesday";
  cout.width( 10 );
```



```
  cout << "wednesday";
  cout << endl;
  cout.width( 10 );
  cout << "1";
  cout.width( 10 );
  cout << "2";
  cout.width( 10 );
  cout << "3";
  cout << endl;
} // end of main()
```

- output:

```
here's a table with lined-up columns:
    monday   tuesday wednesday
         1         2         3
```


- another example:

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

int main() {
  const int A = 5;
  const double B = 3.4568;
  double C;
  cout << "output using fixed precision, 2 decimal places:\n";
  cout.setf( ios::fixed );
  cout.precision( 2 );
  cout << "B=" << B << endl;
  cout << "output using width=10, left justified:\n";
  cout.setf( ios::left );
  cout.width( 10 );
  cout << "B=" << B << endl;
  cout << "output using width=10, right justified:\n";
  cout.setf( ios::right );
```



```
  cout.width( 10 );
  cout << "B=" << B << endl;
  cout << "you have to repeat the formatting if you want the same thing again:\n"
  C = sin( B );
  cout.setf( ios::right );
  cout.width( 10 );
  cout << "C=" << C << endl;
} // end of main()
```

- output:

```
output using fixed precision, 2 decimal places:
B=3.46
output using width=10, left justified:
B=        3.46
output using width=10, right justified:
        B=3.46
you have to repeat the formatting if you want the same thing again:
        C=-0.31
```