Selection of LOOP constructs:

- **For Loop**: Calculation that is repeated a fixed number of times using a value that is changed by equal amounts each time through the loop.

- **Do-While Loop**: The loop boby will be executed at least once.

- **While Loop**: If there are circumstances for which the loop body should not be executed at all.

Nested Loop

Example:

```
// Program that output the times table

#include <iostream.h>

void main ( ) {
    int i,          // Outer loop counter
        j;          // Inner loop counter


      for ( i = 1; i <= 10; i++ ) {
           for ( j = 1; j <= 10; j++ )
                cout << i*j << ' ';
           cout << endl;
      }
}
```

Common Programming errors:

```
while ( balance  !=  0.0 ) {
        balance = balance - amount;
 }
```

- balance may not equal to zero due to **numerical inaccuracies**.

```
while ( power <= 1000 ) {
        cout << "Next power of N is "
                  << Power << endl ;
        power = power * n;
}
```

- Be sure to initialize to 0 a variable used for accumulating a sum.
- Be sure to initialize to 1 a variable used for accumulating a product.

Multiple Selection: Switch Statement

- Syntax

```
switch (<selector expression>) {
  case <label1> : <sequence of statements>;
                   break;
  case <label2> : <sequence of statements>;
                   break;
        •
        •
        •
  case <labeln> : <sequence of statements>;
                   break;
  default :       <sequence of statements>;
}
```

- Meaning
    - o evaluate selector expression
    - o match case label
    - o execute sequence of statements of matching label
    - o if ***break*** encountered, go to end of switch statement
    - o otherwise continue execution

Switch Statement (Cont'd)

- Example

```
switch (watts) {
  case 25 :
    lifespan = 2500;
    break;
  case 60 :
    lifespan = 1000;
    break;
  case 75 :
    lifespan = 750;
    break;
  default :
    lifespan = 0;
} // end switch
```

- Multiple labels

```
switch (watts) {
  case 25 :
    lifespan = 2500;
    break;
  case 40 :
  case 60 :
    lifespan = 1000;
    break;
  case 75 :
    lifespan = 750;
    break;
  default :
    lifespan = 0;
} // end switch
```

Switch Statement (Cont'd)

- Example

```
switch ( testscore / 10 ) {
  case 10 :
  case 9 :
    grade = 'A';  break;
  case 8 :
    grade = 'B';  break;
  case 7 :
    grade = 'C';  break;
  case 6 :
    grade = 'D';  break;
  default :
    grade = 'F';
} // end switch
```

Top-Down Design

Start with the most abstract formulation of a problem and work down to more detailed subproblems.

- Functional decomposition:
  - Decompose a program into functionally separate pieces from the general to the specific in a top-down fashion.
- Abstraction through use of subprograms:
  - Most languages allow the use of both internal and external subprograms. External subprograms are usually grouped into specialized libraries.
  - Subprograms allow abstraction in programming by separating usage (what?) from implementation (how?).
  - Subprograms facilitate reuse of program code.

Functions

- **Function Prototype**

```
<type> <function identifier>(<type list>);
```

```
<type>, <type>, •••
```

- **Function Definition**

```
<type> <function identifier>(<parameters list>)
  {
    <local declarations>
    <sequences of statements>
  }
```

```
<type> <identifier>, •••
```

- **Function Call**

```
<function name>(<argument list>)
```

  - result of function call is a value of type <type>
  - arguments match parameters
  - **void** function - a function that doesn’t return a value

There is a one-to-one correspondence between the <u>arguments</u> in a function call and the <u>parameters</u> in the corresponding function definition.

Arguments Passing

- ***formal argument***
    - o arguments declared in the argument list of a function

- formal arguments are referred to in the body a function

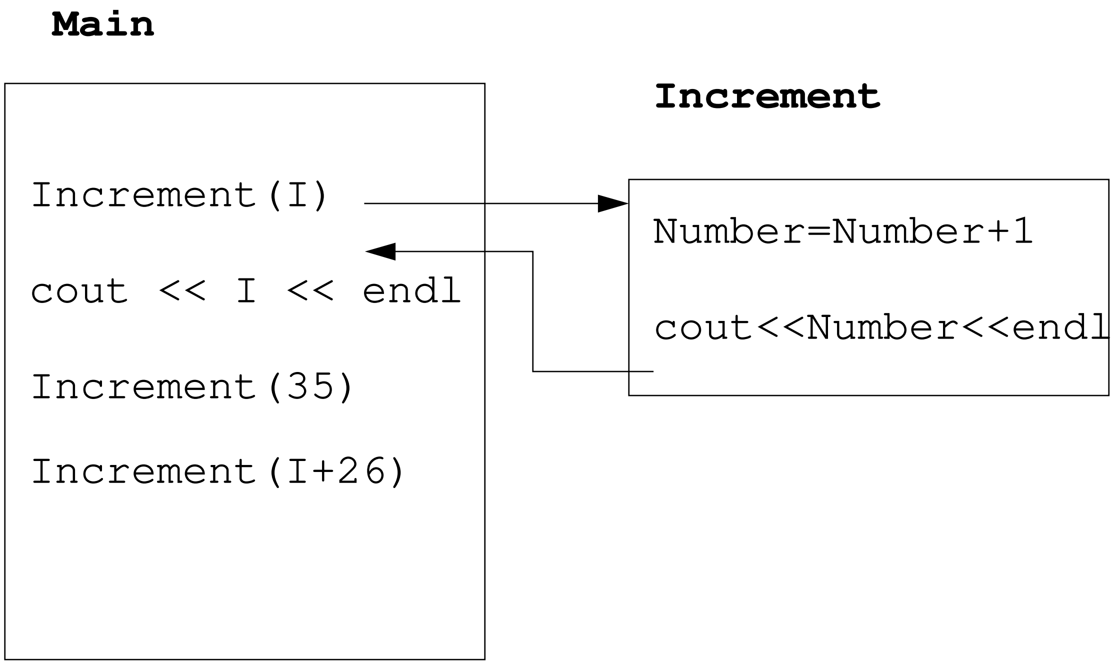
- a function call has to supply a variable, a value or an expression (***actual argument***) for each of the formal argument

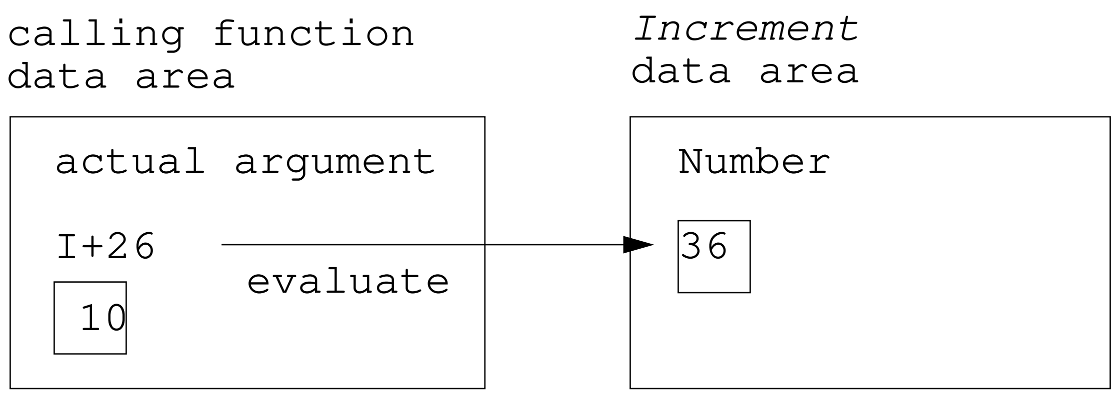
- **argument passing**
    - ***o pass-by-value***
        - each actual argument (a variable, a value or an expression) is ***evaluated*** to a *value*
        - value is ***copied*** to data area of function
        - function manipulates ***local*** copies only
    - *o charateristics*
        - function ***never*** accesses *actual argument*
        - changes are ***not*** reflected in actual argument
        - function terminates, local values are lost

Pass-by-V alue

- Example:

```
#include <iostream.h>

void Increment(int Number) {
    Number = Number + 1;
    cout << Number << endl;
  }

void main() {
  int I = 10;

  //actual argument is a variable
  Increment(I);
  cout << I << endl;

  //actual argument is a value
  Increment(35);

  //actual argument is an expression
  Increment(I+26);
}
```


Main
Increment
Increment(I)
cout << I << endl
Increment(35)
Increment(I+26)
Number=Number+1
cout<<Number<<endl


calling function data area
Increment data area
actual argument
I+26
10
evaluate
Number
36