Benefits of Subprograms

- modularity

- procedural abstraction, implementation hiding

- libraries
Subprograms

\(<\text{name}>)(<\text{parameters}>)\)

**Formal** parameters: names used in the declaration/definition

**Actual** parameters: data at the time of call

**Issues:**

- nesting
- nature of parameters
- nature of returned value
- scope rules
Functions as parameters

#include <iostream>

int x;

int foo (int);

int applytoA (int (*)(int), int [], int);

void main()

{
  int y;
  int A[5] = {2, 4, 6, 8, 10};
  x = 10;
  x = foo(x);
  y = applytoA(foo, A, 5);
  cout << y << "", " << A[1]
       << "\n" ;}
int foo (int y)
{ return (x + y);
}

int applytoA (int (*f) (int), int A[], int size)
{
    int i;
    for (i = 0; i < size; i++) A[i] = f(A[i]);
    return A[0];
}
Activation record

Data needed for activating a procedure

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Control link</td>
<td></td>
</tr>
<tr>
<td>Saved machine state</td>
<td></td>
</tr>
<tr>
<td>Returned value</td>
<td></td>
</tr>
<tr>
<td>Actual parameters</td>
<td></td>
</tr>
<tr>
<td>Local variables</td>
<td></td>
</tr>
<tr>
<td>Temporary values</td>
<td></td>
</tr>
</tbody>
</table>
What are the output values?

```c
#include <stream.h>

void foo (int, int[], int);

main()
{
    int y;
    int bb[5] = {0,1,2,3,4};
    int x = 10;
    foo(x, bb, 5);
    cout << x << ", " << bb[0]
        << "\n" ;

    void foo (int y, int A[], int size)
    {
        y = y + 1;
        A[0] = A[0] + 10;
    }
```
Calls by value and by reference

- call by value: formal parameter corresponds to the \textit{value} of the actual parameter

- call by reference: formal parameter is a synonym for the \textit{location} of the actual parameter
$$l$$-values and $$r$$-values

- $$l$$-value: location of the named data object
  
  Corresponds to the left-hand side of an assignment statement

- $$r$$-value: value of the data object
  
  Corresponds to the right-hand side of an assignment statement

The assignment

$$A = B$$

means

“Assign to the $$l$$-value of A the $$r$$-value of B”
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>05834700</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>A[0]</td>
<td>05834800</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>A[2]</td>
<td>05834802</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>A[3]</td>
<td>05834803</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
Call by value

The $r$-values of the actual parameters are copied into locations specified by the $l$-values of the formal parameters

- one-way communication
- provides input to the called procedure
- protects the actual parameters from change
Sample program: call by value

void swap(int, int);

int main()
{
    int i = 3;
    swap(i, A[i]);
    cout << i << "", " << A[3]
        << "\n";
    return 0;
}

void swap(int x, int y)
{
    int z;
    z = x;
    x = y;
    y = z;
}

will output 3 and 0
Call by reference

The $l$-values of the formal parameters are set equal to the $l$-values of the actual parameters

- Two-way communication
- Provides both input to, and output from the called procedure
- No protection for formal or actual parameters
Sample program: call by reference

```c
void swap(int&, int&);

int main()
{
  int A[5] = {2,3,1,0,4};
  int i = 3;
  swap(i, A[i]);
  return 0;}

void swap(int& x, int& y)
{
  int z;
  z = x;
  x = y;
  y = z;
}

will output 0 and 3
```
Call by name

1. The literal texts of the actual parameters replaces every occurrence of the corresponding formal parameters. (Variables mentioned in these expressions are global to the called procedure.)

2. If necessary, local variables are given new names not appearing elsewhere in the program

- One or two-way communication. If an actual parameter has no l-value, then assignment to the corresponding formal parameter is disallowed

- Delayed evaluation of actual parameters - by need only

- No protection for formal or actual parameters
Sample program: call by name

in C++-style notation

```cpp
void swap(int ‘by-name’, int ‘by-name’);

int main()
{ int A[5] = {2,3,1,0,4};
  int i = 3;
  swap(i, A[i]);
  cout << i << "", " << A[3]
  << "\n" ;
  return 0;}

void swap(int x, int y)
{ int z;
  z = x;
  x = y;
  y = z;
}

will output 0 and 0
```
void swap(int ‘by-name’, int ‘by-name’);

int main()
{ int A[5] = {2,3,1,0,4};
  int i = 3;
  swap(i, A[i]);
  // in effect,
  //   z = i;
  //   i = A[i];
  //   A[i] = z;
      << "\n" ;
  return 0}

void swap(int x, int y)
{ int z;
  z = x;
  x = y;
  y = z;  
}
Macros in C++

#define swap(x,y) {int z; z = x; x = y; y = z;}

int main()
{
    int A[5] = {2,3,1,0,4};
    int i = 3;
    swap(i,A[i]);
    cout << i << " , " << A[3]
         << " \n" ;
    return 0;}

Call by value-result

1. The \( l \)-values of the actual parameters are saved, one for each of the formal parameters.

2. The \( r \)-values of the actual parameters are copied into locations specified by the \( l \)-values of the formal parameters.

3. On return, The \( r \)-values of the formal parameters are copied into locations specified by the saved \( l \)-values of the actual parameters.
Sample program: call by value-result

in C++-style notation

```cpp
void unsafe(int 'by value-result');

int a;

int main()
{
    a = 1;
    unsafe(a);
    cout << a
        << "\n";
    return 0;}

void unsafe(int x)
{
    x = 2;
    a = 0;
}

will output 2
```
void unsafe(int&);

int a;

int main()
{
    a = 1;
    unsafe(a);
    cout << a << "\n";
    return 0;}

void unsafe(int& x)
{
    int z; // a new identifier
    z = x;
    // -----------------------------
    z = 2; // replacing x with z
    a = 0; //
    // -----------------------------
    x = z;
}
Scope

*binding*: mapping of identifiers to data items

*referencing environment*: bindings in effect (at a given point)

*Scope*: textual region of the program in which a binding is active
Lexical and Dynamic scope

Lexical scope: bindings can be determined at compile time

The “current” binding for a name corresponds to the closest lexically enclosing declaration.

Dynamic scope: bindings depend on the flow of execution at run time

The “current” binding for a name is the most recently encountered one during execution.
#include <iostream.h>

int x = 2;

int addx (int);

void test (void);

int main()
{
    test();
    cout << x << "\n" ;
    return 0;}

int addx (int z)
{return (x + z);}

void test (void)
{int x = 10;
    cout << addx(1000) << "\n \n"; }

#!/usr/local/bin/perl
$x = 2;
@test;
print $x, "\n";

sub addx
    {local ($z);
        $z = pop @_;
        return ($x + $z);
    }

sub test
    {local $x;
        $x = 10;
        print &addx(1000), "\n \n";
    };

CSI 311