"People think that computer science is the art of geniuses but the actual reality is the opposite, just many people doing things that build on each other, like a wall of mini stones."
- Donald Knuth

CSE341
Programming Languages

Lecture 7 – November 2021

Procedures

© 2013-2021 Yakup Genç

Slides are taken from C. Li & W. He

1

Procedures vs. Functions

- Function:
  - no side effect
  - return a value
  - Function call: expression
- Procedure:
  - side effect, executed for it
  - no return value
  - Procedure call: statement
- No clear distinction made in most languages
  - C/C++: void
  - Ada/FORTRAN/Pascal: procedure/function

November 2021   CSE341 Lecture 7   2

2

Syntax

- Terminology:
  - body
  - specification interface
    - name
    - type of return value
    - parameters (names and types)

```
                int f(int y);    //declaration

int f(int y) {     int f(int y){    //definition
    int x;             int x;
    x=y+1;             x=y+1;
    return x;          return x;
}                  }
```

November 2021   CSE341 Lecture 7   3

3

Procedure Call

```
Caller:              Callee:
...                  int f(int y){
f(a);                    int x;
...                      if (y==0) return 0;
                         x=y+1;
                         return x;
                     }
```

- Control transferred from caller to callee, at procedure call
- Transferred back to caller when execution reaches the end of body
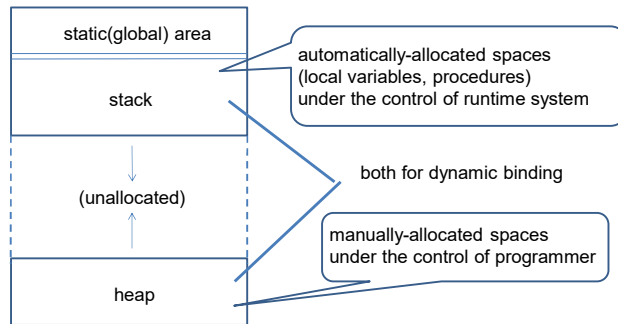- Can **return** early

November 2021   CSE341 Lecture 7   4

4

Environment

- Environment: binding from names to their attributes

static(global) area
stack
(unallocated)
heap

automatically-allocated spaces (local variables, procedures) under the control of runtime system

both for dynamic binding

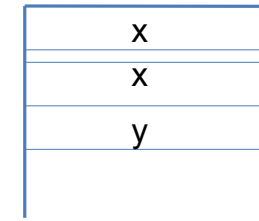manually-allocated spaces under the control of programmer

5

Activation Record for Nested Blocks

- Activation record: memory allocated for the local objects of a block
  - Entering a block: activation record allocated
  - Exit from inner block to surrounding block: activation record released

```
int x; //global
{
   int x,y;
   x = y*10;
   {
      int i;
      i = x/2;
   }
}
```
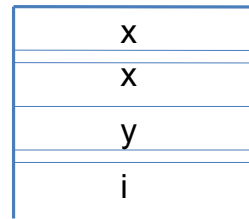
x
x
y

6

Activation Record for Nested Blocks

```
int x; //global
{
   int x,y;
   x = y*10;
   {
      int i;
      i = x/2;
   }
}
```

x: nonlocal variable, in the surrounding activation record
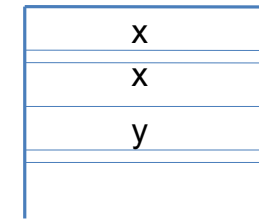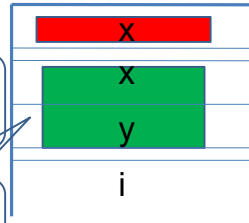
x
x
y
i

7

Activation Record for Procedures

```
int x; //global
void B(void) {
   int i;
   i = x/2;
}
void A(void) {
   int x,y;
   x = y*10;
   B();
}
main() {
   A();
   return 0;
}
```

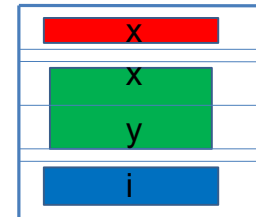| x |
| --- |
| x |
| y |

8

2

Slide 9

Activation Record for Procedures

```
int x; //global
void B(void) {
  int i;
  i = x/2;    ⬅
}
void A(void) {
  int x,y;
  x = y*10;
  B();
}
main() {
  A();
  return 0;
}
```

x: global variable in defining environment

Need to retain information in calling environment

(stack: x (red), x (green), y (green), i)

November 2021   CSE341 Lecture 7   9

9

Slide 10

Activation Record for Procedures

```
int x; //global
void B(void) {
  int i;
  i = x/2;    ⬅
}
void A(void) {
  int x,y;
  x = y*10;
  B();
}
main() {
  A();
  return 0;
}
```

i: local variable in called environment

x: global variable in defining environment

x,y: local variable in calling environment

(stack: x (red), x (green), y (green), i (blue))

November 2021   CSE341 Lecture 7   10

10

Slide 11

Activation Record for Procedures

```
int x; //global
void B(void) {
  int i;
  i = x/2;
}
void A(void) {
  int x,y;
  x = y*10;
  B();
}
main() {
  A();
  return 0;
}
```

Can only access global variables in defining environment

No direct access to the local variables in the calling environment
(Need to communicate through parameters)

(stack: x (red), x (green), y (green), i (blue))
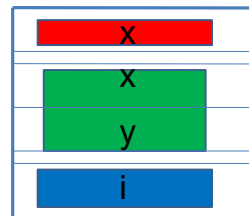
November 2021   CSE341 Lecture 7   11

11

Slide 12

Procedure Call

- Caller: Callee:

```
...                 int f(int a){
f(i);                 ...;
...                   ...a...;
                      ...;
                    }
```

actual parameter / argument

formal parameter / argument

Parameter Passing Mechanisms:

- When and how to evaluate parameters
- How actual parameter values are passed to formal parameters
- How formal parameter values are passed back to actual parameters

November 2021   CSE341 Lecture 7   12

12

3

Parameter Passing Mechanisms

- Pass/Call by Value
- Pass/Call by Reference
- Pass/Call by Value-Result
- Pass/Call by Name

13

Example

- What is the result?

```
void swap(int a, int b) {
    int temp;
    temp = a;
    a = b;
    b = temp;
}
main(){
    int i=1, j=2;
    swap(i,j);
    printf("i=%d, j=%d\n", i, j);
}
```

- It depends…

14

Pass by Value

- Caller: Callee:

```
…                   int f(int a){
f(i);                   ...a...;
…                   }
```


- Most common one
- Replace formal parameters by the values of actual parameters
- Actual parameters: No change
- Formal parameters: Local variables (C, C++, Java, Pascal)
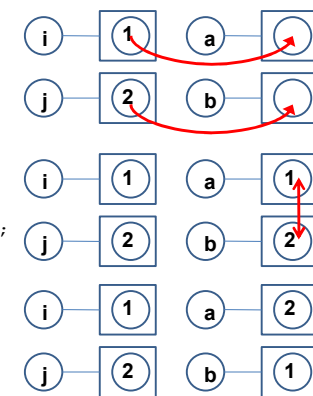
15

Example: Pass By Value

```
void swap(int a, int b) {
    int temp;
    temp = a;
    a = b;
    b = temp;
}
main(){
    int i=1, j=2;
    swap(i,j);
    printf("i=%d, j=%d\n", i, j);
}
```

16

4

Are these Pass-by-Value?

- C:

```
void f(int *p) { *p = 0; }

void f(int a[]) { a[0]=0; }
```

- Java:

```
void f(Vector v) { v.removeAll(); }
```
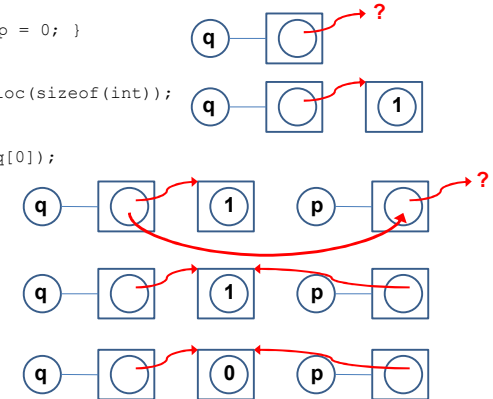
Yes!

17

Pass-by-Value: Pointers

- C:

```
void f(int *p) { *p = 0; }
main() {
   int *q;
   q = (int *) malloc(sizeof(int));
   *q = 1;
   f(q);
   printf("%d\n", q[0]);
}
```

18

Pass-by-Value: Pointers

- C:

```
void f(int *p) { p = (int *) malloc(sizeof(int)); *p = 0; }
main() {
   int *q;
   q = (int *) malloc(sizeof(int));
   *q = 1;
   f(q);
   printf("%d\n", q[0]);
}
```

- What happens here?

19

Pass-by-Value: Arrays

- C:

```
void f(int p[]) { p[0] = 0;}
main() {
   int q[10];
   q[0]=1;
   f(q);
   printf("%d\n", q[0]);
}
```

- What happens here?

20

Pass-by-Value: Arrays

- C:

```
void f(int p[]) { p=(int *) malloc(sizeof(int)); p[0] = 0; }
main() {
  int q[10];
  q[0]=1;
  f(q);
  printf("%d\n", q[0]);
}
```

- What happens here?

21

Pass-by-Value: Java Objects

- Java:

```
void f(Vector v) { v.removeAll(); }

main() {
  Vector vec;
  vec.addElement(new Integer(1));
  f(vec);
  System.out.println(vec.size());
}
```

- What happens here?

22

Pass-by-Value: Java Objects

- Java:

```
void f(Vector v) { v = new Vector(); v.removeAll(); }

main() {
  Vector vec;
  vec.addElement(new Integer(1));
  f(vec);
  System.out.println(vec.size());
}
```

- What happens here?

23

Pass by Reference

- Caller: Callee:
  …   int f(int a){
  f(i);   ...a...;
  …   }

- Formal parameters become **alias** of actual parameters
- Actual parameters: changed by changes to formal parameters
- Examples:
  - Fortran: the only parameter passing mechanism
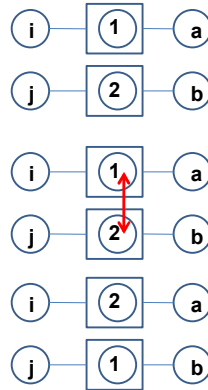 - C++ (reference type, `&`) /Pascal (`var`)

24

6

Example: Pass By Reference

C++ syntax. Not valid in C

```
void swap(int &a, int &b) {
   int temp;
   temp = a;
   a = b;
   b = temp;
}
main(){
   int i=1, j=2;
   swap(i,j);
   printf("i=%d, j=%d\n", i, j);
}
```

i — 1 — a
j — 2 — b

i — 1 — a
j — 2 — b

i — 2 — a
j — 1 — b

25

Pass-by-Reference: How to mimic it in C?

C:

```
void f(int *p) { *p = 0; }
main() {
   int q;
   q = 1;
   f(&q);
   printf("%d\n", q);
}
```

- It is really pass-by-value. Why?

26

It is really pass-by-value

C:

```
void f(int *p) { p = (int *) malloc(sizeof(int)); *p = 0; }
main() {
   int q;
   q = 1;
   f(&q);
   printf("%d\n", q);
}
```

27

Pass-by-Reference: C++ Constant Reference

C++:
```
void f(const int & p) {
   int a = p;
   p = 0;
}
main(){
   int q;
   q = 1;
   f(q);
   printf("%d\n", q);
}
```

const int &p
Error: expression must be a modifiable lvalue

- What happens here?

28

7

Pass-by-Reference: C++ Reference-to-Pointer

C++:
```
void f(int * &p) { *p = 0; }
main(){
    int *q;
    int a[10];
    a[0]=1;
    q=a;
    f(q);
    printf("%d, %d\n", q[0], a[0]);
}
```

0 0

- What happens here?

29

Pass-by-Reference: C++ Reference-to-Pointer

C++:
```
void f(int * &p) { p = new int; *p = 0; }
main(){
    int *q;
    int a[10];
    a[0]=1;
    q=a;
    f(q);
    printf("%d, %d\n", q[0], a[0]);
}
```

0 1

- What happens here?

30

Pass-by-Reference: C++ Reference-to-Array

C++:
```
void f(int (&p)[10]) { p[0]=0; }
main(){
    int *q;
    int a[10];
    a[0]=1;
    q = a;
    f(a);
    printf("%d, %d\n", q[0], a[0]);
}
```
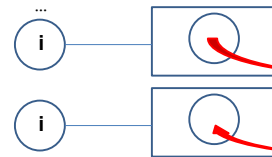
0 0

- What happens here?

31

Pass by Value-Result
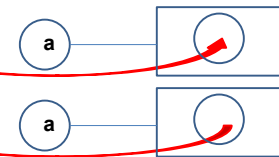
- Caller: Callee:
```
…                    int f(int a){
f(i);                    ...a...;
…                    }
```


- Combination of Pass-by-Value and Pass-by-Reference (Pass-by-Reference without aliasing)
- Replace formal parameters by the values of actual parameters
- Value of formal parameters are copied back to actual parameters
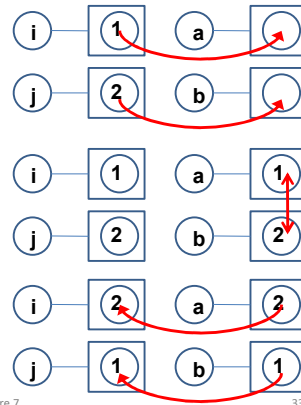
32

8

Example: Pass By Value-Result

```
void swap(int a, int b) {
   int temp;
   temp = a;
   a = b;
   b = temp;
}
main(){
   int i=1, j=2;
   swap(i,j);
   printf("i=%d, j=%d\n", i, j);
}
```
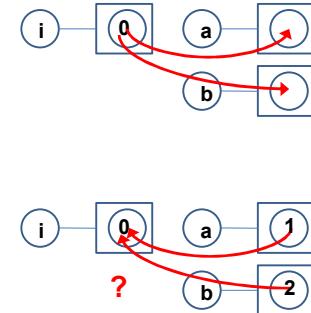
33

Unspecified Issues

```
void f(int a, int b) {
   a = 1;
   b = 2;
}
main(){
   int i=0;
   f(i,i);
   printf("i=%d\n", i);
}
```

?

34

Pass by Name

- Caller: Callee:
  …   `int f(int a){`
  `f(i);`   `...a...;`
  …   `}`

- Actual parameters only evaluated when they are needed
- The same parameter can be evaluated multiple times
- Evaluated in calling environment
- Callee can change the values of variables used in the argument expression and hence change the expression's value
- Essentially equivalent to normal order evaluation
- Example:
  - Algol 60
  - Not adopted by any major languages due to implementation difficulty

35

Evaluation Strategy Revisited

- Strict Evaluation
  - the arguments to a function are always evaluated completely before the function is applied
  - eager evaluation
- Non-strict Evaluation
  - the arguments to a function are not evaluated unless they are actually used in the evaluation of the function body
  - short-circuit & lazy evaluation

36

9

Strict Evaluation Revisited

- **Applicative order**: the arguments of a function are evaluated from left to right
- **Call by value**: the argument expression is evaluated, and the resulting value is bound to the corresponding variable in the function
- **Call by reference**: a function receives an implicit reference to a variable used as argument
- **Call by sharing** (or object): differing from call-by-reference in that assignments to function arguments within the function are not visible to the caller
- **Call by value-result** (or copy-restore): a special case of call-by-reference where the provided reference is unique to the caller (Fortran & in multiprocessing context)

37

Non-strict Evaluation Revisited

- **Normal order** (or leftmost outermost): the outermost reducible expression is always reduced, applying functions before evaluating function arguments
- **Call by name**: the arguments to a function are not evaluated before the function is called, they are substituted (capture-avoiding) directly into the function body and then left to be evaluated whenever they appear in the function
- **Call by need**: a memoized version of call-by-name where, if the function argument is evaluated, that value is stored for subsequent uses. In pure functional programming, this produces the same results as call-by-name; when the function argument is used two or more times, call-by-need is almost always faster.
- **Call by macro extension**: similar to call-by-name, but uses textual substitution rather than capture-avoiding substitution

Memoization is an optimization technique used to speed up programs by having function calls avoid repeating the calculation of results for previously processed inputs

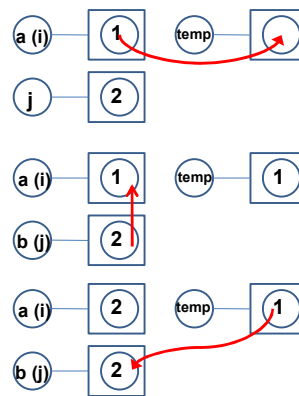38

Example: Pass By Name

```
void swap(int a, int b) {
   int temp;
   temp = a;
   a = b;
   b = temp;
}
main(){
   int i=1, j=2;
   swap(i,j);
   printf("i=%d, j=%d\n", i, j);
}
```
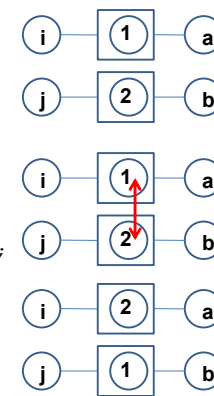
39

Example: Pass By Reference

```
void swap(int &a, int &b) {
   int temp;
   temp = a;
   a = b;
   b = temp;
}
main(){
   int i=1, j=2;
   swap(i,j);
   printf("i=%d, j=%d\n", i, j);
}
```

40

Pass-by-Name: Side Effects

```
int p[3]={3,2,1};
int i;

void swap(int a, int b) {
  int temp;
  temp = a;
  a = b;
  b = temp;
}
main(){
  i = 1;
  swap(i, p[i]);
  printf("%d, %d\n", i, p[i]);
}
```

```
// call to swap(x,y)
temp = x;
x = y;
y = temp;


// call to swap(i,p[i])
temp = i;
i = p[i];
p[i] = temp;

before call …
i=1 and x[1]=3
after call …
i=3 and x[3]=1
```

• **What happens here?**

41

Example: What's the use?

```
double sum(int j, int s, int e, double Ej) {
  double t;
  t = 0;
  for (j=s; j<=e; ) {
    t = t + Ej;
  }
  return t;
}
main(){
  int i=10;
  printf("%f\n", sum(i, 0, 20, x[i]*i);
}
```

42

Comparisons

• **Call by Value**
  – Efficient. No additional level of indirection.
  – Less flexible and less efficient without pointer.
    • (array, struct, union as parameters)
• **Call by Reference**
  – Require one additional level of indirection (explicit dereferencing)
  – If a parameter is not variable (e.g., constant), a memory space must be allocated for it, in order to get a reference.
  – Easiest to implement.
• **Call by Value-Result**
  – You may not want to change actual parameter values when facing exceptions.
• **Call by Name**
  – Lazy evaluation
  – Difficult to implement

43