CSE 3302
Programming Languages
Lecture 5: Control

(based on the slides by Chengkai Li)
Leonidas Fegaras
University of Texas at Arlington
Control

- Control:
  what gets executed, when, and in what order

- Abstractions of control:
  - Expression
  - Statement
  - Exception handling
  - Procedures and functions
Expression vs. Statement

• Difference in purity
  – pure = no side effects
  – given the same input, you always get the same output

• Expressions:
  • no side effects
  • return a value

• Statements:
  • side effects
  • no returned value

• Functional languages aim at achieving this purity
  – no statements - no side effects

• No clear-cut in most languages
Expressions

- Constructed recursively:
  - Basic expressions (literal, identifiers)
  - Operators, functions, special symbols
- Various number of operands:
  - unary, binary, ternary operators
- Operators & functions: equivalent concepts
  - \((3+4)\times 5\) (infix notation)
  - \((\text{multiply}(\text{add}(3,4),5))\) (Ada, prefix notation)
  - \((\ast\ (+\ 3\ 4)\ 5)\) (LISP, prefix notation)
Expressions and Side Effects

- **Side Effects:**
  - changes to memory, input/output operations (read/print)
  - Side effects can be undesirable
    - To understand a function, you need to understand how it interacts with the environment and the other functions that change the same state
  - A program *without* side effects produces the same output for the same input
    - \( x = y \) implies \( f(x) = f(y) \)
    - boring?

- **Expressions:**
  - No side effects: The order of evaluating subexpressions doesn’t matter (mathematical forms)
  - Side effects: Order matters
    - Let \( x = 1 \)
    - What is the result of \((x = x + 1) + (x = x * 2)\)?
Applicative Order Evaluation

- Also known as strict evaluation
- Evaluates the operands first, then applies operators
  - bottom-up evaluation
  - subexpressions are always evaluated, no matter whether they are needed

```
  *
 / 
+  -
|  |
3 4 5 6
```
Order Matters

C:
int x=1;
int f(void) {
  x=x+1;
  return x;
}

main(){
  printf("%d\n", x + f());
  return 0;
}

Java:

```
class example
{
  static int x = 1;
  public static int f()
  {
    x = x+1;
    return x;
  }

  public static void main(String[] args)
  {
    System.out.println(x+f());
  }
}
```

Many languages don’t specify the order
- C: usually right-to-left
- Java: always left-to-right, but should not rely on that
Predictable Side Effects

• Assignments in expressions
  \[ x = (y = z) \] (= is a right-associative operator)

• Using \texttt{x++} and \texttt{++x}

```c
int x = 1;
int f(void) {
    return x++;
}
main() {
    printf("%d\n", x + f());
    return 0;
}
```
The Sequence Operator

- (expr1, expr2, ..., exprn)
  - Left to right (this is indeed specified in C)
  - The return value is exprn

```c
x=1;
y=2;
x = (x=x+1, y++, x+y);
printf("%d\n", x);
```
Non-Strict Evaluation

- Evaluating an expression without necessarily evaluating all the subexpressions

- short-circuit Boolean expression
  - e1 or e2

- if-expression, case-expression
Short-Circuit Evaluation

• if (false and x) ... if (true or x)...
  – No need to evaluate x

• What is it good for?
  – if (i <=lastindex and a[i]>=x) ... 
  – if (p != NULL and p->next==q) ... 

• Ada: allows both short-circuit and non short-circuit
  – if (x /= 0) and then (y/x > 2) then ...
  – if (x /= 0) and (y/x > 2) then ...
  – if (ptr = null) or else (ptr.x = 0) then ...
  – if (ptr = null) or (ptr.x = 0) then ...
if-expression

- if (test-exp, then-exp, else-exp)
  
  ternary operator
  
  - test-exp is always evaluated first
  - either then-exp or else-exp are evaluated, not both

  - if e1 then e2 else e3        (ML)
  - e1 ? e2 : e3               (C)

- What about the if-statement?


- ML:

```haskell
case color of
  red => "R"
|  blue => "B"
|  green => "G"
|  _  => "AnyColor"
```


Loops

• While loop:
  – while (e) S;
  – Same as:

    loop: if (not e) goto exit;
    S;
    goto loop;
    exit:

• The GOTO controversy
  – Java prohibits it

• Do-while loop:
  – do S while (e);
  – Same as: { S; while (e) S; }
Loops

• For-loop:
  – for (e1; e2; e3) S;
  – Same as: { e1; while (e2) { S; e3 } }
  – eg, for (int i=0; i<10; i++) a[i]++
  – Most languages enforce restrictions on for-loops

• A break statement can be used inside a loop to exit the loop
  – Labeled break

• A continue statement can be used to jump to the next loop step
  – Labeled continue
Exception Handling

• An implicit control mechanism
• An exception is any unexpected or infrequent event
  – Typically is an error event
  – Raise (or throw) an exception: the point where the exception occurs
  – Handle (or catch) an exception: code to be executed when a particular exception occurs
• Try-catch blocks in C++
  struct Trouble {} t;
  try { code that may raise an exception using 'throw' }
  catch (Trouble t) { code that handles the Trouble exception }
  catch (...) { code that handles the rest of all exceptions }
• throw t;
• Propagating exceptions and call/stack unwinding
• Most languages provide predefined exceptions
Normal Order Evaluation

• Also known as lazy evaluation
  – Operation evaluated before the operands are evaluated;
  – Operands evaluated only when necessary

• int double (int x) { return x+x; }
int square (int x) { return x*x; }

Applicative order evaluation:
• square(double(2)) = square(4) = 16

Normal order evaluation:
• square(double(2)) = double(2)*double(2) = (2+2)*(2+2)
What is it good for?

- You can define if-then-else, case, etc as functions

\[(p!=\text{NULL}) \ ? \ p->\text{next} : \text{NULL}\]

```c
int if_exp(bool x, int y, int z)
{
    if (x)
        return y;
    else
        return z;
}
if_exp(p!=\text{NULL}, p->\text{next}, \text{NULL});
```
Examples

• Call by Name (Algol60)
• Macro

#define swap(a, b) {int t; t = a; a = b; b = t;}

What is the problem here?
Unhygienic Macros

- Call by Name (Algol60)
- Macro

```c
#define swap(a, b) {int t; t = a; a = b; b = t;}

main (){
    int t=2;
    int s=5;
    swap(s,t);
}
```

```c
#define DOUBLE(x) {x+x;}

main() {
    int a;
    a = DOUBLE(get_int());
    printf("a=%d\n", a);
}
```

```c
main (){
    int t=2;
    int s=5;
    {int t; t = s; s = t; t = t;}
}
```

```c
main() {
    int a;
    a = get_int()+get_int();
    printf("a=%d\n", a);
}
```
Procedures vs. Functions

• Function:
  • no side effects
  • returns a value
  • a function call is an expression
• Procedure:
  • side effects
  • no returned value
  • a procedure call is a statement
• No clear distinction made in most languages
  – C/C++: a procedure is a function that returns void
  – Ada/FORTRAN/Pascal: procedure/function
Syntax

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

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

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

```java
int f(int y) {
    int x;
    x=y+1;
    return x;
}
```
Procedure Call

• Caller:
  ...
  f(a);
  ...

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

• At the call: control transferred from the caller to the callee
• Transferred back to the caller when execution reaches the end of the body
• Can \texttt{return} early
Runtime Environment

- Environment: binding from names to their attributes

![Diagram showing runtime environment with static (global) area, stack, heap, and unallocated space]

- **Static (global) area**
- **Stack**
- **Heap**

  - Automatically-allocated space for local variables; under the control of runtime system
  - Manually-allocated space; under the control of programmer
Activation Records for Nested Blocks

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

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

x: nonlocal variable, in the surrounding activation record
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;
}
Activation Records for Procedures

```c
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 the defining environment
- Need to retain information in the calling environment
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 the called environment

x: global variable in the defining environment

x, y: local variables in the calling environment
Activation Records for Procedures

```c
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 the defining environment

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

• Caller:
  ...
  f(i);
  ...

Callee:
  int f(int a) {
    ...
    ...a...;
    ...
  }

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
Parameter Passing Mechanisms

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

• What is the result?

```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);
}
```

• It depends...
Pass by Value

- Caller:
  
  ```
  ... 
  f(i); 
  ...
  ```

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

- This is the most common form
- Replace formal parameters by the values of actual parameters
- Actual parameters: No change
- Formal parameters: Local variables (C, C++, Java, Pascal)
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);
}
Are these Pass-by-Value?

• C:
  ```c
  void f(int *p) { *p = 0; }
  void f(int a[]) { a[0]=0; }
  ```

• Java:
  ```java
  void f(Vector v) { v.removeAll(); }
  ```
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]);
}
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?
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?
Pass-by-Value: Arrays

• C:

```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?
Pass-by-Value: Java Objects

- **Java:**
  ```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?**
Pass-by-Value: Java Objects

- Java:
  ```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?
Pass by Reference

- Caller:
  
  ... 
  f(i);
  ...

- Callee:

  int f(int a){
    ...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)
Example: Pass By Reference

C++ syntax. Not valid in C

```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);
}
```
Pass-by-Reference: How to mimic it in C?

• 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?
It is really pass-by-value

- C:

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

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

• What happens here?
Pass-by-Reference: C++ Reference-to-Pointer

- **C++:**
  ```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]);
  }
  ```

- **What happens here?**
Pass-by-Reference: C++ Reference-to-Pointer

- C++:
  ```cpp
  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]);
  }
  ```

- What happens here?
• **C++:**

```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]);
}
```

• **What happens here?**
Pass by Value-Result

- **Caller:**
  
  ```
  ...;
  f(i);
  ...
  ```

- **Callee:**
  
  ```
  int f(int a){
    ...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
Example: Pass By Value-Result

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

- **Caller:**
  ```
  ... 
  f(i);
  ...
  ```

- **Callee:**
  ```
  int f(int a) {
      ...a...;
  }
  ```

- The actual parameters are only evaluated when they are needed
- The same parameter can be evaluated multiple times
- They are evaluated in the calling environment
- Essentially equivalent to normal order evaluation
- Example:
  - Algol 60
  - Not adopted by any major languages due to implementation difficulty
Example: Pass By Name

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

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

• What happens here?
Some Variants

• Pass by Name
  – Evaluated at every use, in the calling environment

• Pass by Need
  – Evaluated once, memorized for future use

• Pass by Text (Macro)
  – Evaluated using the called environment.

• All belong to non-strict evaluation (lazy evaluation)
Comparisons

• **Call by Value**
  – Efficient; No additional level of indirection
  – Less flexible and less efficient without pointers
    • array, struct, union as parameters

• **Call by Reference**
  – Requires 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
Heap-Allocated Data

- Dynamic memory management
- The simplest heap allocator that does not reclaim storage:

```c
char heap[heap_size];

int end_of_heap = 0;

void* malloc ( int size ) {
    void* loc = (void*) &heap[end_of_heap];
    end_of_heap += size;
    return loc;
}
```
With a Free List

- Need to recycle the dynamically allocated data that are not used
- This is done using free in C or delete in C++
- Need to link all deallocated data in the heap into a list
- Initially, the free list contains only one element that covers the entire heap (i.e., it's heap_size bytes long)

```c
typedef struct Header { struct Header *next; int size; } Header;
Header* free_list;
```

- `free` simply puts the recycled cell at the beginning of the free list:

  ```c
  void free ( void* x ) {
      if ("size of *x" <= sizeof(Header))
          return;
      ((Header*) x)->next = free_list;
      ((Header*) x)->size = "size of *x";
      free_list = (Header*) x;
  }
  ```
Malloc

- malloc first searches the free list to find a cell large enough to fit the given number of bytes. If it finds one, it gets a chunk out of the cell leaving the rest untouched:

```c
void* malloc ( int size ) {
    Header* prev = free_list;
    for (Header* r = free_list; r != 0; prev = r, r = r->next)
        if (r->size > size + sizeof(Header))
            { 
                Header* new_r = (Header*) (((char*) r) + size);
                new_r->next = r->next;
                new_r->size = r->size;
                if (prev==free_list)
                    free_list = new_r;
                else prev->next = new_r;
                return (void*) r;  }

    void* loc = (void*) &heap[end_of_heap];
    end_of_heap += size;
    return loc;  }
```
Reference Counting

• Keeping track of pointer assignments
• If more than one objects point to a dynamically allocated object, then the latter object should be deleted only if all objects that point to it do not need it any more
• you need to keep track of how many pointers are pointing to each object
• Used to be popular for OO languages like C++
• Note: this is not automatic garbage collection because the programmer again is responsible of putting counters to every object to count references
• This method is easy to implement for languages like C++ where you can redefine the assignment operator dest=source, when both dest and source are pointers to data
Reference Counting (cont.)

- Instead of using C++ for a pointer to an object C, we use Ref<O>, where the template Ref provides reference counting to C:

template< class T >
class Ref {
  private:
    int count;
    T* pointer;
  void MayDelete () {
    if (count==0) delete pointer;
  };
  void Copy ( const Ref &sp ) {
    ++sp.count;
    count = sp.count;
    pointer = sp.pointer;
  };
  public:
    Ref ( T* ptr = 0 ) : count(1), pointer(ptr) {};
    Ref ( const Ref &sp ) { Copy(sp); };
    ~Ref () { MayDelete(); };
    T* operator-> () { return pointer; };
    Ref& operator= ( const Ref &sp ) {
      if (this != &sp) {
        count--;
        MayDelete();
        Copy(sp);
      }
      return *this;
    };
};