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)*5
  – multiply( add(3,4), 5)
    • “+”(3,4,5) (Ada, prefix notation)
    • (“*” (+ 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

Order Matters

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

Java:
```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 $x++$ and $++x$
```java
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`

```
x=1;
y=2;
x = (x=x+1, y++, x+y);
printf("%d\n", x);
```
**case-expression**

- **ML:**
  ```
  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*

```c
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)

Examples

- Call by Name (Algol60)
- Macro

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

What is the problem here?

What is it good for?

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

```c
(p!=NULL) ? p->next : NULL
```

```c
int if_exp(bool x, int y, int z)
{ if (x)
    return y;
  else
    return z;
}
```

```c
if_exp(p!=NULL, p->next, NULL);
```

Unhygienic Macros

- Call by Name (Algol60)
- Macro

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

```c
int t=2;
int s=5;
swap(s,t);
```

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

```c
#define DOUBLE(x) (x+x)
```

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

```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)

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

Procedure Call

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

Runtime Environment

- Environment: binding from names to their attributes

```plaintext
static (global) area

stack

(unallocated)

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;
}
```

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;
}
```
Activation Records for Procedures

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

- `x`: global variable in the defining environment
- `i`: local variable in the called environment
- `x,y`: local variables in the calling environment

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:**
  
  ```c
  ...
  f(i);
  ...
  ```

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

- Callee:
  ```
  i
  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)

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);
}
```

Are these Pass-by-Value?

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

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

- C++:
  ```
  void f(int a[]){ a[0]=0; }
  ```
**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]);
  }

- 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: 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=(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.remove(); }
  
  public class Main {
      public static void main(String[] args) {
          Vector vec = new Vector();
          vec.addElement(new Integer(1));
          f(vec);
          System.out.println(vec.size());
      }
  }
  ```

- What happens here?

Pass by Reference

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

- Callee:
  ```java
  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, & in C, 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;
}
```
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?

---

Pass-by-Reference: C++ Constant Reference

- 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++:
  ```cpp
  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
define f(int * a) { 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?

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

- C++:
  ```cpp
define f(int (*a)[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:
  ```cpp
  f(i);
  ...
  ...
  ```

- Callee:
  ```cpp
  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

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

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);
  ...
}

• 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

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-Name: Side Effects

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)
  ```
  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:
  ```
  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<C>, where the template Ref provides reference counting to C:

```cpp
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;
        };
```