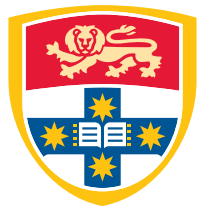
Introduction to Programming (Adv)

School of Computer Science, University of Sydney



Copyright Warning

Control flow

Control flow

Jump first

Linear code execution. Line 1, 2, 3, 4...

Show the compounded monthly interest calculated for a whole year

```
float earned, new_savings;
char *endp = NULL;
float savings = strtof(argv[1], &endp );
float irate = strtof(argv[2], &endp ); // no error checking!

earned = savings * irate / 12.0f;
new_savings = savings + earned;
printf("Savings are $%.2f. Monthly interest $%.2f\n", new_savings
    , earned);

savings = new_savings;
earned = savings * irate / 12.0f;
new_savings = savings + earned;
printf("Savings are $%.2f. Monthly interest $%.2f\n", new_savings
    , earned);

...
```

Execution context

Where are we?

```
7  float earned, new_savings;
8  char *endp = NULL;
9  float savings = strtof(argv[1], &endp );
10 float irate = strtof(argv[2], &endp ); // no error checking!
11
12 earned = savings * irate / 12.0f;
13 new_savings = savings + earned;
14 printf("Savings: $%.2f. Monthly interest: $%.2f\n", new_savings,
       earned);
15
16 savings = new_savings;
17 earned = savings * irate / 12.0f;
18 new_savings = savings + earned;
19 printf("Savings: $%.2f. Monthly interest: $%.2f\n", new_savings,
       earned);
```

PC = Program counter (rip) and more memory!

The GNU debugger

Starting and stepping through the program

```
gcc -g -o program linear.c
```

```
gdb program
run
```

Execution:

```
run
```

```
breakpoint 12
info locals
list
print $pc
next
print $pc
next
print $pc
next
print $pc
continue
```

Memory layout

Variables mapped onto registers by compiler

rbp/ebp - Base pointer (Address of beginning of stack memory)

rsp/esp - Stack pointer (Address of current stack position)

```
info frame
info stack
x/20x $rsp
x/20x $rbp
print new_savings
print &new_savings
x/1f &new_savings
```

Branching

Simple branching instructions

```
3  int main(int argc, char **argv) {
4      if ( argc > 3 ) {
5          argc = 754;
6          return 1;
7      } else {
8          argc = argc + 109;
9      }
10
11     return 0;
12 }
```

```
gcc -S branch1.c
```

Branching

Comparison and jump

Program counter automatically changes

```
5   main:
6   .LFB0:
7       .cfi_startproc
8       pushq   %rbp
9       .cfi_def_cfa_offset 16
10      .cfi_offset 6, -16
11      movq    %rsp, %rbp
12      .cfi_def_cfa_register 6
13      movl    %edi, -4(%rbp)
14      movq    %rsi, -16(%rbp)
15      cmpl    $3, -4(%rbp)
16      jle .L2
17      movl    $754, -4(%rbp)
18      movl    $1, %eax
19      jmp .L3
```

```
20  .L2:
21      addl    $109, -4(%rbp)
22      movl    $0, %eax
23  .L3:
24      popq    %rbp
25      .cfi_def_cfa 7, 8
26      ret
27      .cfi_endproc
```

```
mov - register <-> memory
add - ALU
cmp - comparison operation
jmp - unconditional jump
je/jne/jle/jge - conditional jump
call,ret - subroutine calls
push/pop - stack operations
```

Same effect

Loops are another case for jump instructions

```
3  int main(int argc, char **argv) {
4      while ( argc < 17 ) {
5          argc++;
6      }
7      return 0;
8  }
```

Same effect (cont.)

```
5  main:
6  .LFB0:
7          .cfi_startproc
8          pushq   %rbp
9          .cfi_def_cfa_offset 16
10         .cfi_offset 6, -16
11         movq    %rsp, %rbp
12         .cfi_def_cfa_register 6
13         movl    %edi, -4(%rbp)
14         movq    %rsi, -16(%rbp)
15         jmp .L2
16 .L3:
17         addl    $1, -4(%rbp)
18 .L2:
19         cmpl    $16, -4(%rbp)
20         jle .L3
21         movl    $0, %eax
22         popq    %rbp
23         .cfi_def_cfa 7, 8
24         ret
25         .cfi_endproc
```

```
3  int main(int argc, char **argv) {
4      while ( argc < 17 ) {
5          argc++;
6      }
7      return 0;
8  }
```

```
mov - register <-> memory
add - ALU
cmp - comparison operation
jmp - unconditional jump
je/jne/jle/jge - conditional jump
call,ret - subroutine calls
push/pop - stack operations
```

Bad branching

The bank will grant bonus interest if X transactions are completed in the month.

```
25  float rate;
26  if (bonus_eligible) {
27      rate = base_rate + bonus_rate;
28  } else {
29      rate = base_rate;
30  }
31
32  earned = savings * rate / 12.0f;
33  new_savings = savings + earned;
34  if (bonus_eligible)
35      printf("Savings: $%.2f. Monthly interest (with bonus): $%.2f\n",
            new_savings, earned);
36  else
37      printf("Savings: $%.2f. Monthly interest: $%.2f\n", new_savings, earned)
            ;
```

More with GDB

Set the stopping conditions

```
breakpoint (line, function)
watch (data breakpoint)

list [<linenumber>]
info locals
print <variable/$register>
continue
backtrace
next (step over, no subroutines)
```

Let's watch the `new_savings`

Functions branch too

Subroutines require more work for making the jump work.

Save the state - PC & registers

```
3  int foo(int x) {
4      int y = x - 1;
5      return y;
6  }
7
8  int main(int argc, char **argv) {
9      while ( argc < 3 ) {
10         argc = foo(argc);
11     }
12     return 0;
13 }
```

Additionally, we have a return value to worry about

call - save return address, set new rsp

Functions branch too (cont.)

```
 5  foo:
 6  .LFB0:
 7      .cfi_startproc
 8      pushq   %rbp
 9      .cfi_def_cfa_offset 16
10      .cfi_offset 6, -16
11      movq    %rsp, %rbp
12      .cfi_def_cfa_register 6
13      movl    %edi, -20(%rbp)
14      movl    -20(%rbp), %eax
15      subl    $1, %eax
16      movl    %eax, -4(%rbp)
17      movl    -4(%rbp), %eax
18      popq    %rbp
19      .cfi_def_cfa 7, 8
20      ret
21      .cfi_endproc
22  .LFE0:
23      .size   foo, .-foo
24      .globl  main
25      .type   main, @function
```

```
26  main:
27  .LFB1:
28      .cfi_startproc
29      pushq   %rbp
30      .cfi_def_cfa_offset 16
31      .cfi_offset 6, -16
32      movq    %rsp, %rbp
33      .cfi_def_cfa_register 6
34      subq    $16, %rsp
35      movl    %edi, -4(%rbp)
36      movq    %rsi, -16(%rbp)
37      jmp .L4
38  .L5:
39      movl    -4(%rbp), %eax
40      movl    %eax, %edi
41      call    foo
42      movl    %eax, -4(%rbp)
43  .L4:
44      cmpl    $2, -4(%rbp)
45      jle .L5
46      movl    $0, %eax
47      leave
48      .cfi_def_cfa 7, 8
49      ret
50      .cfi_endproc
```

Savings with functions

Identify the useful tasks for this program

What is a good interface to operate on inputs, what are the outputs, and what about errors?

```
 8  static int parse_arguments(char **argv, float *savings, float *base_rate,
 9                             float *bonus_rate, bool *bonus_eligible);
10
11  static void compound_interest_month(
12      float *savings, const float base_rate,
13      const float bonus_rate, const bool bonus_eligible,
14      float *earned);
15
16  static void print_statement(
17      float savings, bool bonus_eligible, float earned);
```

Savings with functions (cont.)

```
73  float savings, base_rate, bonus_rate;
74  bool bonus_eligible;
75
76  int error = parse_arguments(argv, &savings, &base_rate,
77                              &bonus_rate, &bonus_eligible);
78  if (0 != error)
79      return error;
80  // post condition valid float and bool values
81
82  // calculate and update savings balance for 12 months
83  int month = 1;
84  while (month <= 12) {
85      float earned;
86      compound_interest_month(&savings, base_rate,
87                              bonus_rate, bonus_eligible, &earned);
88      print_statement(savings, bonus_eligible, earned);
89      month++;
90  }
91  // post condition: savings variable updated based on interest rates
92  // post condition: output to stdout
```

Savings with functions (cont.)

```
73  static void compound_interest_month(
74      float *savings, const float base_rate,
75      const float bonus_rate, const bool bonus_eligible,
76      float *earned)
77  {
78      float rate;
79      if (bonus_eligible) {
80          rate = base_rate + bonus_rate;
81      } else {
82          rate = base_rate;
83      }
84
85      *earned = *savings * rate / 12.0f;
86      float new_savings = *savings + *earned;
87
88      *savings = new_savings;
89  }
```

Navigate with GDB and functions

Set the stopping conditions

```
help running
step (step to next line, with subroutines)
finish (end of current stack frame)
```

Structured conditional local jumps

switch - equivalence for categorical options selection

```
if (x == value1) {
    statement1;
} else if (x == value2) {
    statement2;
} else if (x == value3) {
    statement3;
} else if (x == value4) {
    statement4;
} else {
    statement_when_x_matches_none;
}
```

```
switch (x) {
    case value1:
        statement1;
        break;
    case value2:
        statement2;
        break;
    case value3:
        statement3;
        break;
    case value4:
        statement4;
        break;
    default:
        statement_when_x_matches_none
            ;
        break;
}
```

Structured conditional local jumps (cont.)

Fall through case statements

```
if (x == value1a || x == value1b) {
    statement1;
} else if (x == value2) {
    statement2;
} else if (x == value3a)
    if (x == value3b || x ==
        value3c) {
        statement3;
    }
} else if (x == value3b || x ==
    value3c) {
    statement3;
} else if (x == value4) {
    statement4;
} else {
    statement_when_x_matches_none;
}
```

```
switch (x) {
    case value1a:
    case value1b:
        statement1;
        break;
    case value2:
        statement2;
        break;
    case value3a:
        statement3a;
    case value3b:
    case value3c:
        statement3;
        break;
    case value4:
        statement4;
        break;
    default:
        statement_when_x_matches_none
            ;
        break;
}
```

break - specifically to escape a loop structure, switch

Unstructured unconditional local jumps

Notice that there are labels within the assembly code. These can also be used in C with the goto statement.

goto considered harmful!

Unstructured unconditional local jumps (cont.)

```
if (conditionA)
{
    statement1;
    if (conditionB)
    {
        statement2;
        if (conditionC)
        {
            goto special_place;
        }
        else
        {
            statement3;
        }
    } else
    {
        statement4;
    }
}
else
{
special_place:
    // executes when !conditionA | (conditionA & conditionC)
    statement 5;
}
```

Unstructured unconditional local jumps (cont.)

goto useful for some cases, but it is generally hard to justify against using structured programming

```
73  static int transfer_money_mediator(
74      int socket_a, int socket_b, int socket_c,
75      char *bsb_a, char *bsb_b, char *bsb_c,
76      char *acc_a, char *acc_b, char *acc_c,
77      float amount, const char *type)
78  {
79      if (-1 == connect(socket_a))
80          goto failed_connect;
81      if (-1 == check_bsb(bsb_a, socket_a);
82          goto failed_accounts;
83      if (-1 == check_acc(acc_a, socket_a);
84          goto failed_accounts;
85      if (-1 == init_transaction(socket_a, type))
86          goto failed_transaction_type_type;
87
88      // same for b, c
89
90      // check mediator (C)
91      if (-1 == confirm_parties(socket_c, bsb_a, bsb_b, acc_a, acc_b))
92          goto failed_parties;
93
94      // check party A
95      if (-1 == confirm_parties(socket_a, bsb_a, bsb_b, acc_a, acc_b))
96          goto failed_parties;
97
```

Unstructured unconditional local jumps (cont.)

```
98      // check party B
99      if (-1 == confirm_parties(socket_b, bsb_a, bsb_b, acc_a, acc_b))
100         goto failed_parties;
101
102     // ...
103
104     // SUCCESS
105     return 0;
106
107 failed_connect:
108     fprintf(stderr, "A party could not connect\n");
109     return 1;
110 failed_accounts:
111     fprintf(stderr, "A party rejected their own account details for the
            transaction\n");
112     return 2;
113 failed_transaction_type:
114     fprintf(stderr, "A party rejected the type of transaction\n");
115     return 3;
116 failed_parties:
117     fprintf(stderr, "A party disagreed with the transaction details\n");
118     return 4;
119
120 }
```

Unstructured unconditional nonlocal jumps in C

`jmp_buf` - A C data structure to store the registers (required for state tracking)

```
setjmp(<jmp_buf> ); // Save register states
```

```
longjmp(<jmp_buf>, 1);
```

zero return value for setjmp, but non-zero return value for setjmp after a longjmp call

Summary

Computer programs operate on an execution context

Control flow is manipulation of the program counter and registers

Structured programming is built on top of the primitives of compare and jump operations.

Functions require special attention for the jump operations to isolate the execution context from the function caller

`if`, `else`, `while`, `do/while`, `switch`, `break`, `continue`, `return` are high level operators in control flow.