THE UNIVERSITY OF SYDNEY

INFO1910 S1 2023 Week 4 Tutorial

Control Flow

A program without a loop and a structured variable isn't worth writing.
- A. J. Perlis, Epigrams on Programming, 1982

cmp & jmp

So far we have carefully tried to avoid presenting problems with indeterminate length programs. Almost all prior tasks could be completed without the use of switches or loops. This week we'll be delving into the guts of how to control a program.

We've previously seen how we can construct a switch from first principles, just using basic operators. Now we'll try to use that switch to manage our control flow.

The program has a notion of a 'state', this tracks what function we are currently executing, where in the function we currently are and what variables are located in the current function. For the moment we will consider two regions of memory; the stack where variables are stored, and program memory where functions containing instructions are stored. As both of these areas are in memory, they are both addressable and these addresses can be stored in pointers.

The most important things to track in this state are the base pointer `bp` and the program counter `pc`. The base pointer indicates the base of the current stack frame. As we have discussed in previous weeks, all variables are referenced in terms of their offset from the current stack frame. The program counter instead points to the current instruction, when the instruction is complete it is incremented to the next instruction.

Our first notion of control flow then requires saving the state of the registers. We can then return to a previous state by setting the current registers to those same values. This is a jump or a `jmp`

To control when these jumps occur, we have the family of `cmp` operations. These operations compare a stored value to some other value, these can then be paired with a conditional jump to only jump if a condition is met.

The following code includes saved states, comparisons and a jump back to a previously saved state.

```c
#include <stdio.h>
#include <setjmp.h>

// Putting this in the global namespace for neatness for the moment
// This stores the current state of the registers
jmp_buf buf;

int main()
{
    int i = 0;

    setjmp(buf); // Save register states

    printf("%d\n", i);
    i++;

    // Ternary Operator, if the condition is met then execute
    // The first section, if it is not met execute the second
    // Here we're putting some garbage operation for the second
    (i < 10) ? longjmp(buf, 1):i--;

    return 0;
}
```

The equivalent assembly (with labels left in, and a few lines strategically removed) for this code is here.

```
main:
.LFB0:
    movl    $0, -12(%ebp)
    subl    $12, %esp
    movl    buf@GOT(%ebx), %eax
    pushl   %eax
    movl    %ebx, -28(%ebp)
    call    _setjmp@PLT
    addl    $16, %esp
    subl    $8, %esp
    pushl   -12(%ebp)
    movl    -28(%ebp), %ebx
    leal    .LC0@GOTOFF(%ebx), %eax
    pushl   %eax
    call    printf@PLT
    addl    $16, %esp
    addl    $1, -12(%ebp)
    cmpl    $9, -12(%ebp)
    jg  .L3
    subl    $8, %esp
```

```
    pushl   $1
    movl    buf@GOT(%ebx), %eax
    pushl   %eax
    call    longjmp@PLT
.L3:
    subl    $1, -12(%ebp)
    movl    $0, %eax
    leal    -8(%ebp), %esp
    popl    %ecx
    ret
```

You can produce this yourself by compiling with the `-S` flag, this particular version of the assembly still retains the labels for unlinked objects. You can see the final product using the `objdump` command with the `-d` or `-D` flags on your final binary file.

Question 1: Pattern Recognition

- Try to locate each line of C code in the assembly (some lines of C code may be multiple lines of assembly).
- What is `-12(%ebp)`?
- What is `jg .L3`
- What is `cmpl $9, -12(%ebp)`?
- What is `.L3`, how does it differ from our longjmp?
- What does the call operator do?

Functions

Having uncovered what our base pointer is, we can now construct a very crude approximation of how function calls must work.

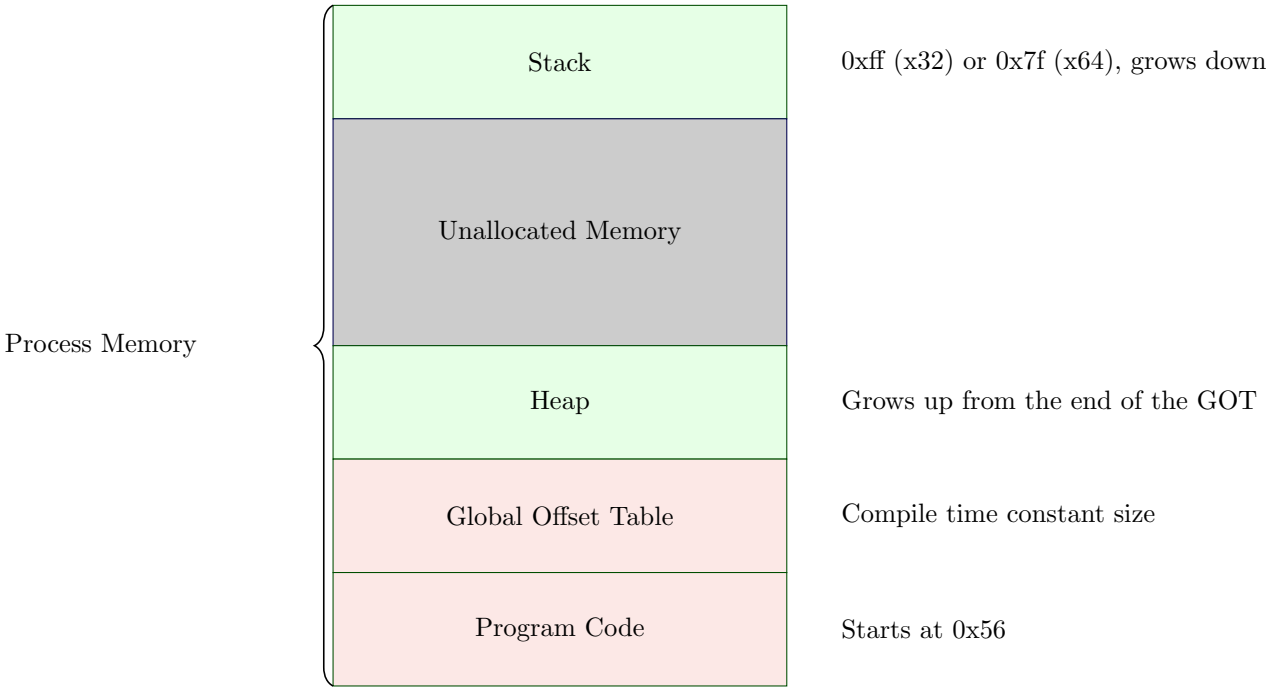

Figure 1: Approximation of standard process memory layout.

When we call a function, a new stack frame is created, the old base pointer and the program counter are saved. The base pointer is then set to base of the new stack frame and the program counter to the start of the stored instructions for the called function. Variables in the new stack frame can then be referenced using the offset from the new value of the base pointer. Variables in the old function are now out of scope and cannot be referenced as their references depend on their offset from the old base pointer.

```c
void foo() {
    int x = 7;
    char y = 42;
    double z = 19.6;
    return 0;
}
```

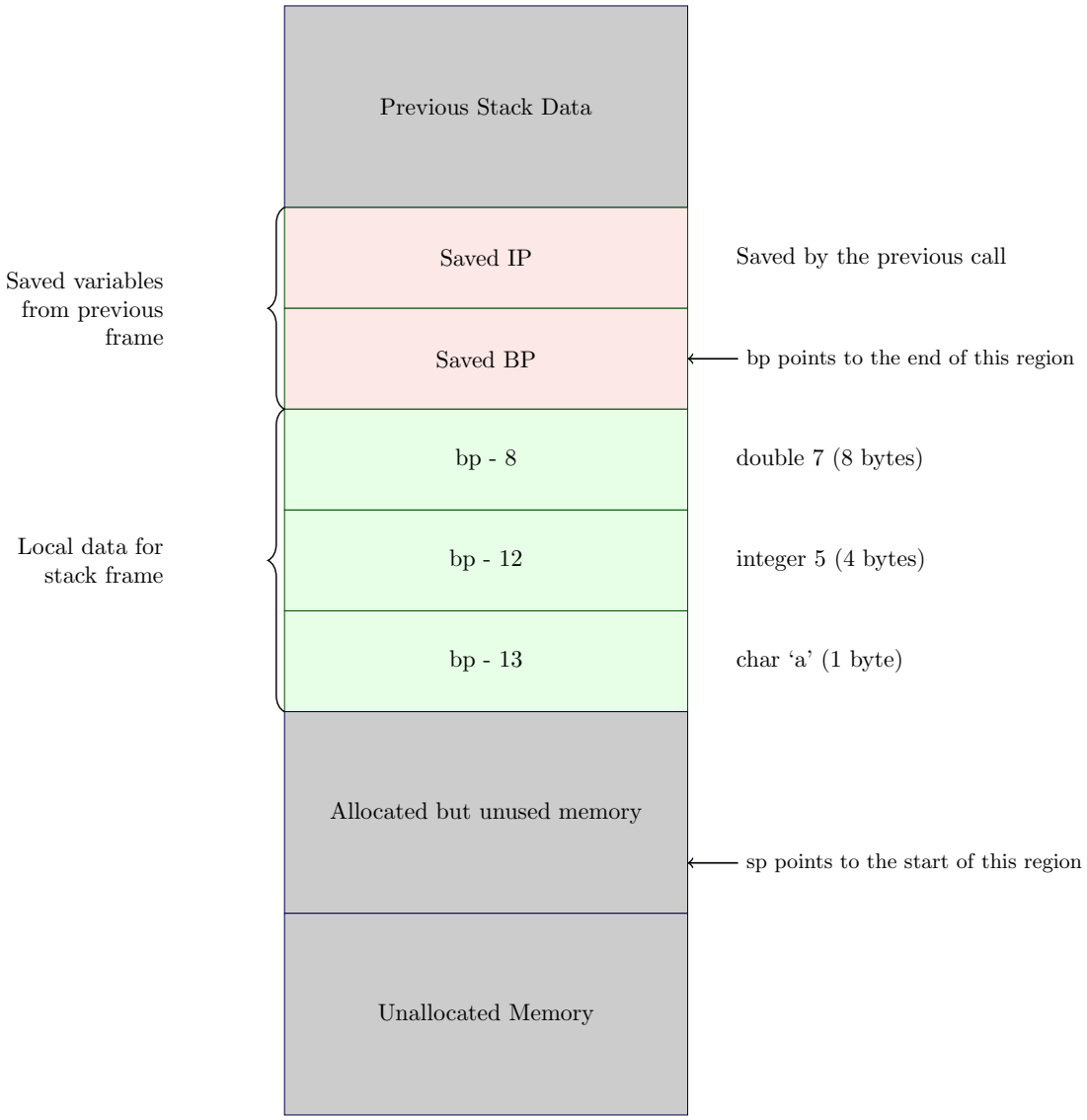

Figure 2: One possible layout of a stack frame of the **foo** function. Note that the order of the variables may not represent their order on the stack.

When the called function ends, we can simply restore the base pointer and program counter to their stored values and continue with the original function. If the called function had a return value, then some section of memory from the original stack frame will be set aside to store the returned values. The size of this memory is determined at compile time by the return type of the function.

It's important to note that the only difference between calling a function, and our previous long jumps is that calling a new function creates a new stack frame to store data in.

Question 2: Stacking up

Given the above description of how function calls occur, what is wrong with the following program?

```c
/*
 * int_doubler
 * Takes an integer value, doubles it
 * :: int x :: The integer to double
 * Returns a pointer to the doubled integer
 */
int* int_doubler(int x);

int main()
{
    int x = 5;
    x = *int_doubler(x);
    printf("%d\n", x);
    return 0;
}

/*
 * int_doubler
 * Takes an integer value, doubles it
 * :: int x :: The integer to double
 * Returns a pointer to the doubled integer
 */
int* int_doubler(int x)
{
    int doubled = x * 2;
    return &doubled;
}
```

if else

`cmp` is somewhat cumbersome, and ternary notation is a pain. Our equivalent statement in C and many other languages is the switch, or the `if` statement.

```c
if (1 == x)
{
    // Do something
}
else if (0 == x)
{
    // Do something else
}
```

```
else
{
    // Another thing
}
```

You can see how this can be constructed, as above, using a series of cmp and jmp operations. Unlike cmp and jmp, the switch can pre-calculate the jump lengths at compile time and perform them at run time. It makes for much neater code than the alternative.

Question 3: Truth in Fiction

Write a simple program that checks for and prints exactly one of the following conditions:

- The number of arguments is greater than four, but not eight
- The number of arguments inputs is less than four, but not two
- The number of arguments inputs is four
- The number of arguments is eight
- The number of arguments is two

Remember how `argv` and `argc` work for this, and me sure to use a makefile!

Question 4: Fib

With your newfound knowledge of switches and functions, write a simple recursive implementation that takes a single command line argument and interprets it as an intger. The program should then find the nth fibonnaci number, where n is the integer representation of the command line argument. You may only use function calls for this problem, you may not use loops.

You may find the `atoi` or `sscanf` functions very useful here.

Examples:

```
./fib 1
1
./fib 3
2
./fib 6
8
```

goto

Now that we have switches, we can ditch the use of ternary operators and jmps with old fashioned 'goto' labels and jumps.

```c
int main()
{
    int i = 0;

    LOOP: // Our Label

    printf("%d\n", i);
    i++;

    if (i < 10)
    {
        goto LOOP; // Jump execution to the label
    }

    return 0;
}
```

We could also construct one with two labels, this is slightly more complicated, but allows us to test the condition upon entering our loop rather than when leaving it.

Unless the question explicitly asks you to, you should now never use goto again. It leads to awful, unreadable code.

Question 5: GOTO GAOL

Use switches and goto statements to construct a program that takes two command line arguments as integers. It then calculates the sum of all integers between these two numbers. Don't cheat with Euler's theorem.

You may find the `atoi` function very useful here.

Question 6: GOTO FIB

Rewrite your Fibonacci program to use goto rather than recursive function calls.

do-while

Some times, you might wish for your loop to execute once before hitting the starting condition, also known as a post-condition loop. This particular sort of loop is vary rarely used, but is included here for completeness.

```c
int i = 0;
do {
    printf("%d\n", i);
    i++;
} while (i < 10);
```

You might notice that is is the same as our earlier single label goto loop, and is hence a bit easier to implement using gotos.

```c
int main()
{
    int i = 0;

    LOOP: // Our Label

    printf("%d\n", i);
    i++;

    if (i < 10)
    {
        goto LOOP; // Jump execution to the label
    }

    return 0;
}
```

while

A common pattern with goto based loops was a two label, pre-condition loop.

```
int main()
{
    int i = 0;

    LOOP_START: // Our Label
    if (i <= 10) // Condition
    {
        goto LOOP_END; // Jump execution to the label
    }

    printf("%d\n", i);
    i++;
    goto LOOP_START:
    LOOP_END: // End Label

    return 0;
}
```

We can take this common construct of a condition followed by a jump from our previous two label goto loop. Here the presentation is quite a bit nicer.

```
while (i <= 10)
{
    printf("%d\n", i);
    i++;
}
```

Question 7: Looping While

Rewrite questions 5 and 6 to use while loops rather than goto statements.

for

A very common pattern with while loops is to loop 'over' some variable. This can be a range of integers, a series of pointers or something else.

You could construct such a loop just using a while:

```c
int i = 0;
while (i < 10)
{
    printf("%d\n", i);
    i++;
}

for (int i = 0; i < 10; i++)
{
    printf("%d\n", i);
}
```

Question 8: Echo again

Using a for loop, read through each command line argument passed using argv and print it to standard output.

Question 9: Strlen Again

Write three different functions that calculate the length of a string, one should use a for loop, one a while and one a do-while. See which loops are better suited to solving this problem.

Question 10: Byte Count

Write a simple program that prints the total number of bytes passed to your command line arguments. Whitespace does not count towards the number of bytes.

```
./byte_count abc 123 456 abc
12
./byte_count
0
./byte_count abc123456abc
12
```

Switch Case

```c
int main(int argc, char** argv)
{
    switch(argc)
    {
        case 1:
            printf("One argument!");
            break;

        case 2:
            printf("Two arguments!");
            break;

        case 3:
            printf("Three arguments!");
            break;

        default:
            printf("Neither one, two or three arguments were passed");
    }

    return 0;
}
```

Question 11: Calculator II

Rewrite last week's calculator program to use a switch.

break, continue

You might have noticed that break syntax from before escapes the current scope. With this and the continue keyword, we have re-introduced very limited goto operations. A break is a jump to the end of the current scope, while a continue is a jump to the start of the current scope.

The following goto statements could be rewritten with breaks and continues:

```
CONTINUE_LABEL:
while (x < 5)
{

    if (x < 3)
    {
        goto CONTINUE_LABEL;
    }

    if (x > 4)
    {
        goto BREAK_LABEL;
    }
}
BREAK_LABEL:
```

And with breaks and continues

```
while (x < 5)
{

    if (x < 3)
    {
        continue;
    }

    if (x > 4)
    {
        break;
    }
}
```

Because of their similariy to goto, use of break and continue is generally frowned upon.

Question 12: Mini-grep

Write a program that emulates a simple version of 'grep' for your command line arguments. The first argument should be the substring to search for, each subsequent string should be a target. You should only return strings that contain the substring, each string should be printed on a new line.

Examples:

```
./not_grep abc abcdef ab aaabc a23b3c
abcdef
aaabc
```

Question 13: strcpy

Write a function strcpy that takes two strings and a position as arguments. We wish to attempt to copy the second string to the nth byte of the first, if space permits.

Take your inputs from standard input. The atoi function might be useful. Position is indexed from 0.

Examples:

```
./strcpy busybee oi 5
busyboi
./strcpy quokka quoll 7
Target is beyond the length of the string!
./strcpy quokka quoll 4
Target string does not have enough space to store the copied string
./strcpy frog toads 0
Target string does not have enough space to store the copied string
```

Question 14: Back to Basics

Rewrite all the above questions using just goto or setjmp and longjmp. Use your newfound arcane knowledge for good and promise never to use a goto in anger, and hopefully not at all.