Return Sequence

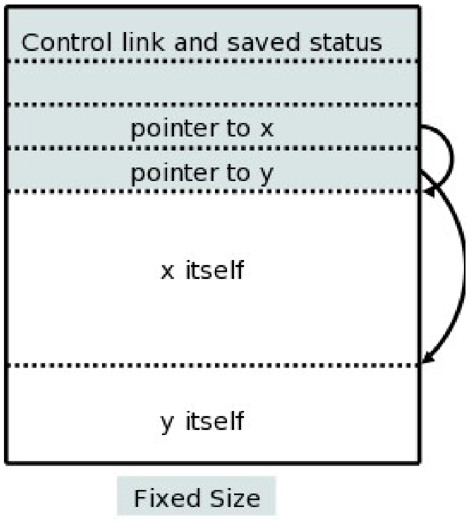
1. The callee stores the return value near the parameters. Note that this address can be **determined by the caller using the old** (soon-to-be-restored) **sp**.
2. The callee **restores sp** and the registers.
3. The callee jumps to the return address.

Note that **varagrs** are supported.

Variable-Length Data on the Stack

There are **two flavors of variable-length data**.

- Data obtained by **malloc/new** have hard to determine lifetimes and are stored in the **heap instead of the stack**.
- Data, such as **arrays with bounds determined by the parameters** are **still stack like** in their lifetimes (if A calls B, these variables of A are allocated before and released after the corresponding variables of B).



It is the **second flavor** that we **wish to allocate on the stack**. The goal is for the (called) procedure to be **able to access these arrays using addresses determinable** at compile time even though the size of the arrays (and hence the location of all but the first element) is **not known until the program is called**, and indeed often differs from one call to the next (even when the two calls correspond to the same source statement). The **solution is to leave room for pointers to the arrays in the AR**. These are **fixed size** and can thus be accessed using **static**

offsets. **Then when the procedure is invoked and the sizes are known, the pointers are filled in and the space allocated**.

A small change caused by storing these variable size items on the stack is that it no longer is obvious where the real top of the stack is **located relative to sp**. Consequently another pointer (call it real-top-of-stack) is also kept. This is used on a call to tell where the new allocation record should begin.

Access to Nonlocal Data on the Stack

As we shall see the ability of **procedure p** to access **data declared outside** of **p** (either declared **globally outside** of all procedures or declared **inside another procedure q**) offers interesting challenges.

1. Data Access without Nested Procedures

In languages like **standard C without nested procedures**, **visible names** are either **local** to the procedure in question or are **declared globally**.

1. For **global names** the address is **known statically at compile time**, providing there is only one source file. If there are multiple source files, the linker knows. In either case **no reference to the activation record is needed**; the addresses are known prior to execution.
2. For **names local** to the **current procedure**, the address needed is in the **AR** at a known-at-compile-time **constant offset** from the **sp**. In the case of variable size arrays, the constant offset refers to a pointer to the actual storage.

2. Issues with Nested Procedures

With nested procedures a **complication** arises. Say **g** is **nested inside f**. So **g** can **refer to names declared in f**. These **names** refer to **objects in the AR for f**; the **difficulty** is finding that **AR when g is executing**. We can't tell at compile time **where the** (most recent) **AR** for **f** will be relative to the current **AR for g** since a **dynamically-determined** (i.e., statically unknown) **number of routines could have been called in the middle.** **Access links are a mechanism to access variables defined in an enclosing procedure**

Parameter Passing

Almost every language has some method for passing parameters to functions and procedures. These mechanisms have **evolved over times**, and there are a number of important differences.

The **two most common methods for parameter passing in modern imperative** languages are **call by value** and **call by reference**. **Call by name** is primarily of historical interest, but is closely related to the way that parameters are handled when macros are expanded, and have the same semantics as lazy evaluation in functional languages. Languages with **input/output parameters** support **call by value-result** which differs subtly from **call by reference**

Techniques used for argument passing in **traditional imperative languages**:

- **call by value**
- **call by result**
- **call by value-result**
- **call by reference**
- **call by name**

Call by value: **copy going into the procedure**. This is the mechanism used by both **C and Java**. Note that this mechanism is used for **passing objects**, where a reference to the objected is passed by value.

Call by result: **copy going out of the procedure**, the formal parameters is copied back into the actual parameters. (Note that this only makes sense if the actual parameter is a variable, or has a l-value, such as an array element.)

Call by value result: **copy going in, and again going out**

Call by reference: **The actual parameters and formal parameters are identified**. The natural mechanism for this is to pass a pointer to the actual parameter, and **indirect through the pointer**

Call by name: **re-evaluate the actual parameter on every use**. For actual parameters that are **simple variables, this is the same as call by reference**. For actual parameters that are **expressions**, the expression is re-evaluated on each access. It should be a runtime error to assign into a formal parameter passed by name, if the actual parameter is an expression.

Call by value is particularly efficient for **small pieces** of data (such integers), since they are trivial to copy, and since access to the formal parameter can be done efficiently.

Call by reference is particularly efficient for **large pieces of data** (such as large **arrays**), since they don't need to be copied. Call by reference also allows a procedure to **manipulate the values of variables of the caller**, such as in a **swap routine.**
FORTRAN uses **call by reference**, early versions of FORTRAN allowed constants to change their values, since they were also passed by reference.
Algol 60 has call by name, call by value

Scheme, Lisp, and Smalltalk use **call-by-value with pointer semantics**
C generally uses call-by-value, although there are inconsistencies in the language. In particular, compare how structures are passed, and how arrays are passed.

An important related concept: <u>aliasing</u>. Two variables are aliased if they refer to the same storage location. A common way that aliasing can arise is using **call by reference.** A formal parameter can be aliased to a nonlocal variable, or two formal parameters can be aliased.

<u>**Examples**</u>

void f(int a, int &b, const int &c);

Parameter **<u>a</u>** is a **value** parameter, **<u>b</u>** is a **reference** parameter, and **<u>c</u>** is a const-reference parameter.

When a parameter is passed by value, a **<u>copy</u>** of the parameter is made. Therefore, **changes made to the formal parameter** by the called function have no effect on the corresponding actual parameter. **For example**:

```
void f(int n) {
   n++;
}

int main() {
   int x = 2;
   f(x);
   cout << x;
}
```

In this example, **<u>f's</u>** parameter is **passed by value**. Therefore, although **<u>f</u>** increments its **formal parameter <u>n</u>**, that has no effect on the actual parameter **<u>x</u>**. The value output by the program is 2 (not 3).

Note that if a **pointer** is **passed by value**, then although the pointer itself is not affected by changes made to the corresponding formal parameter, the **object pointed by the pointed can be changed**. **For example**:

```
void f(int *p) {
   *p = 5;
    p = NULL;
}

int main() {
   int x=2;
   int *q = &x;

   f(q);

   // here, x == 5, but q != NULL
}
```

In this example, **f's** parameter is passed by value. Therefore, the assignment **p = NULL**; in **f** has no effect on variable **q** in main (since f was passed a copy of q, not q itself). However, the assignment *p = 5:in f, **does** change the value pointed to by q. To understand why, consider what happens when the example program runs:

After executing the two statements:

```
   int x=2;
   int *q = &x;
```

memory looks like this:

```
    +---+
 x: | 2 | <--+
    +---+    |
             |
    +---+    |
 q: | --|----+
    +---+
```

Now function **f** is called; the value of **q** (which is the address of x) is copied into a new location named p:

```
       +---+
    x: | 2 | <--+  <--+
       +---+    |     |
                |     |
       +---+    |     |
    q: | --|----+     |
       +---+          |
                      |
       +---+          |
    p: | --|----------+
       +---+
```

Executing the two statements in f:

```
  *p = 5;
   p = NULL;
```

causes the values of x (the thing pointed to by p) and p to be changed:

```
       +---+
    x: | 5 | <--+
       +---+    |
                |
       +---+    |
    q: | --|----+
       +---+

       +----+
    p: |NULL|
       +----+
```

However, note that q is NOT affected.

When a **parameter is passed by reference**, conceptually, the actual parameter itself is passed (and just given a **new name** -- the name of the corresponding formal parameter). Therefore, any changes made to the **formal parameter** do affect the actual parameter. **For example**:

```
void f(int &n) {
   n++;
}

int main() {
   int x = 2;
   f(x);
   cout << x;
}
```

In this example, **f's** parameter is **passed by reference**. Therefore, the assignment to **n** in **f** is actually changing variable **x**, so the output of this program is **3**.

Another common use of reference parameters is for a function that swaps two values:

```
void swap( int &j, int &k ) {
   int tmp = j;
   j = k;
  k = tmp;
}
```

This is useful, for example, in sorting an array, when it is often necessary to swap two array elements. The following code swaps the jth and kth elements of array **A**:

```
swap(A[j], A[k]);
```