More Variable Scope

Scope: Local variables

- The scope of a variable is the portion of the code in which the variable is accessible
- In C, local variables are declared inside a block (hence, *internal* to a function)
  - The scope of these variables is the reminder of the block
- ...or in a function declaration
  - The scope of these variables is the remainder of the function

Scope: Global variables

- In C, global variables are defined outside of (*external* to) blocks and functions
  - Could be in header (.h) files, but shouldn't be!
- The scope of a global variable is the file in which it is declared
  - Can extend the scope of the global variable to other files by using an *extern* declaration

```
extern int g_x;
```

Tells the compiler that **`int g_x`** is global and defined in another file

No memory allocated for g_x here

or use **`flib.h`** header file

Global variables

`flib.c`

```
int g_x;
int g_y = 0;
```

`f1.c`

```
extern int g_x;
extern int g_y;
```

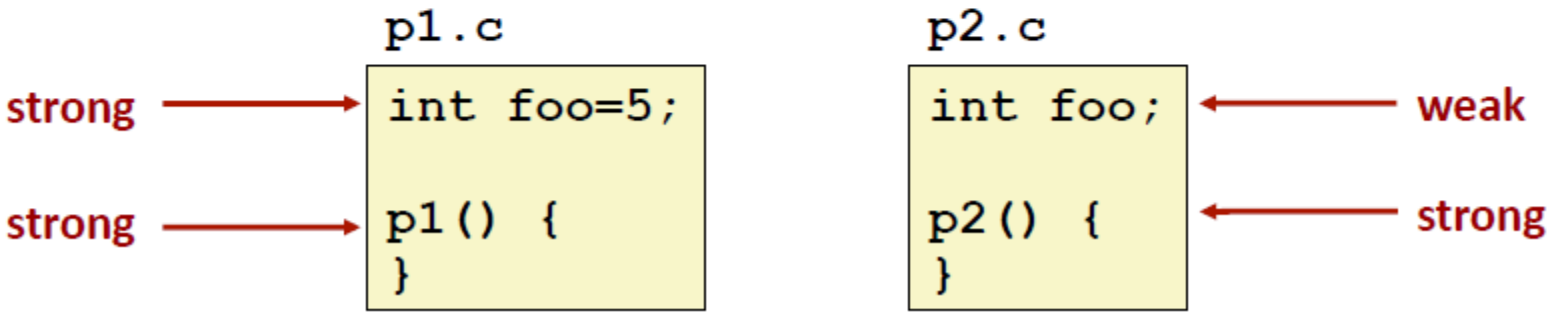
`f2.c`

```
extern int g_x;
extern int g_y;
```

- Exactly one declaration of a global variable omits the word **`extern`**
  - This is where the variable is initialized (optional)
- Declarations in all other files "must" use **`extern`**
  - There are exceptions – don't ask...!

Strong and Weak Symbols

- **Program symbols are either strong or weak**
  - ***Strong***: procedures and initialized globals
  - ***Weak***: uninitialized globals



Linker's Symbol Rules

- **Rule 1: Multiple strong symbols are not allowed**
  - Each item can be defined only once
  - Otherwise: Linker error
- **Rule 2: Given a strong symbol and multiple weak symbol, choose the strong symbol**
  - References to the weak symbol resolve to the strong symbol
- **Rule 3: If there are multiple weak symbols, pick an arbitrary one**
  - Can override this with `gcc -fno-common`

Linker Puzzles

```
int x;
p1() {}
```

```
p1() {}
```

Link time error: two strong symbols (**p1**)

```
int x;
p1() {}
```

```
int x;
p2() {}
```

References to **x** will refer to the same uninitialized int. Is this what you really want?

```
int x;
int y;
p1() {}
```

```
double x;
p2() {}
```

Writes to **x** in **p2** might overwrite **y**!
Evil!

```
int x=7;
int y=5;
p1() {}
```

```
double x;
p2() {}
```

Writes to **x** in **p2** will overwrite **y**!
Nasty!

```
int x=7;
p1() {}
```

```
int x;
p2() {}
```

References to **x** will refer to the same initialized variable.

Nightmare scenario: two identical weak structs, compiled by different compilers with different alignment rules.

Parameter Passing Style

Parameter Passing Techniques

- Pass by Value
- Pass by Reference
- Pass by Pointer

Example: Swapping two values

```
int main()
{
    int  n1, n2;

    cout << "Enter two numbers: " << flush;
    cin >> n1 >> n2;
    if( n1 > n2 )
       Swap( n1, n2 );

    cout << "Sorted order: " << n1 << ", " << n2 << endl;

    return 0;
}
```

How to write `swap()`?

Pass by Value

```
void Swap( int n1, int n2 )
{
    int temp = n1;
    n1 = n2;
    n2 = temp;
}
```

Pass by Reference

```
void Swap( int& n1, int& n2 )
{
    int temp = n1;
    n1 = n2;
    n2 = temp;
}
```

Pass by Pointer

```
void Swap( int* n1, int* n2 )
{
    int temp = *n1;
    *n1 = *n2;
    *n2 = temp;
}
```

Swap without using temp variable

```
void Swap( int &n1, int &n2) {
     n1 = n1^n2;
     n2 = n1^n2;
     n1 = n1^n2;
}
```

```
void Swap( int &n1, int &n2) {
     n2 = n1^n2;
     n1 = n1^n2;
     n2 = n1^n2;
}
```

More Functions

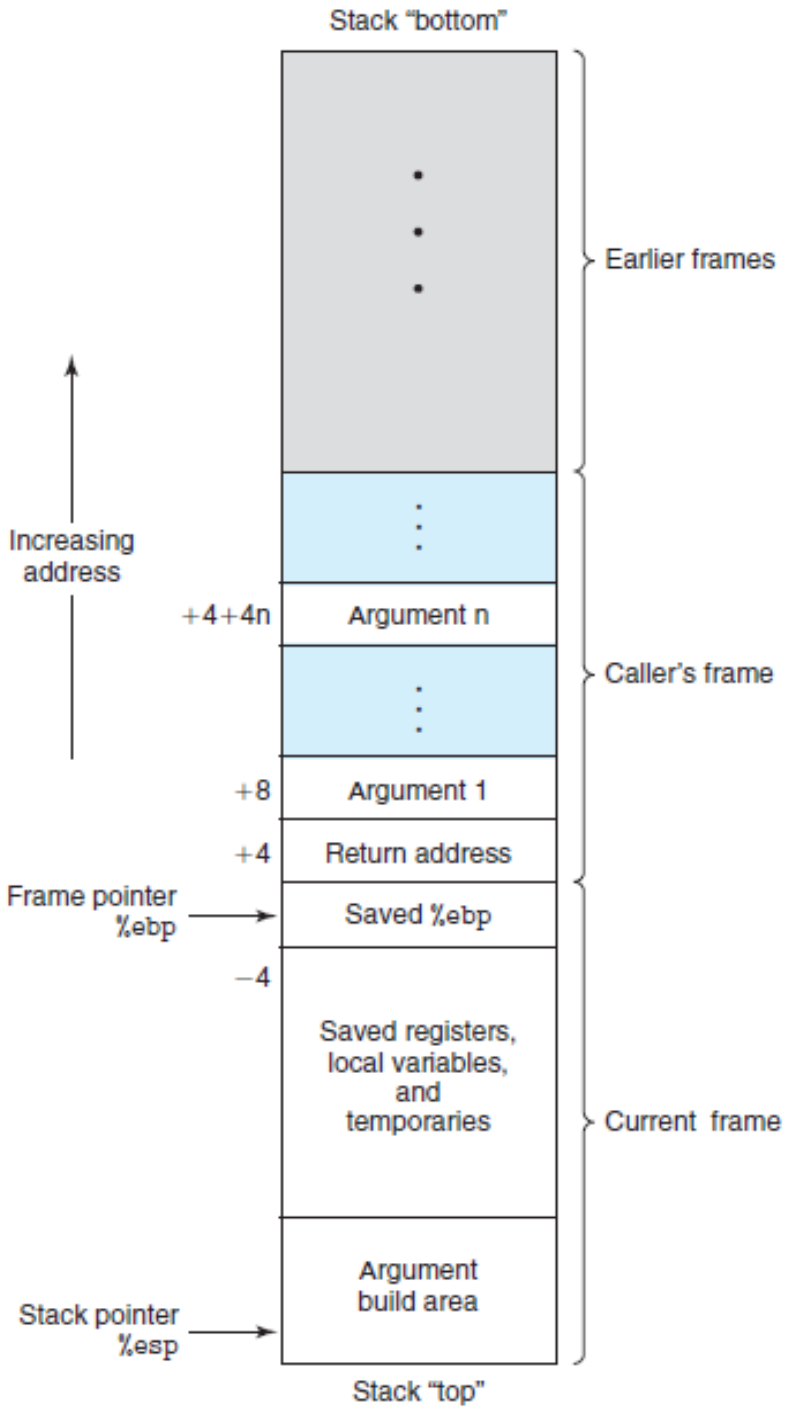
Stack Frame structure

- A procedure call involves passing both data (in the form of procedure parameters and return values) and control from one part of a program to another.
- In addition, it must allocate space for the local variables of the procedure on entry and deallocate them on exit.
- Most machines, including IA32, provide only simple instructions for transferring control to and from procedures.
- The passing of data and the allocation and deallocation of local variables is handled by manipulating the program stack.

Stack Frame structure

- IA32 programs make use of the program stack to support procedure calls.
- The machine uses the stack to pass procedure arguments, to store return information, to save registers for later restoration, and for local storage.
- The portion of the stack allocated for a single procedure call is called a stack frame.

Figure 3.21
Stack frame structure. The stack is used for passing arguments, for storing return information, for saving registers, and for local storage.



```
1   int swap_add(int *xp, int *yp)
2   {
3       int x = *xp;
4       int y = *yp;
5
6       *xp = y;
7       *yp = x;
8       return x + y;
9   }
10
11  int caller()
12  {
13      int arg1 = 534;
14      int arg2 = 1057;
15      int sum = swap_add(&arg1, &arg2);
16      int diff = arg1 - arg2;
17
18      return sum * diff;
19  }
```

Figure 3.23 Example of procedure definition and call.

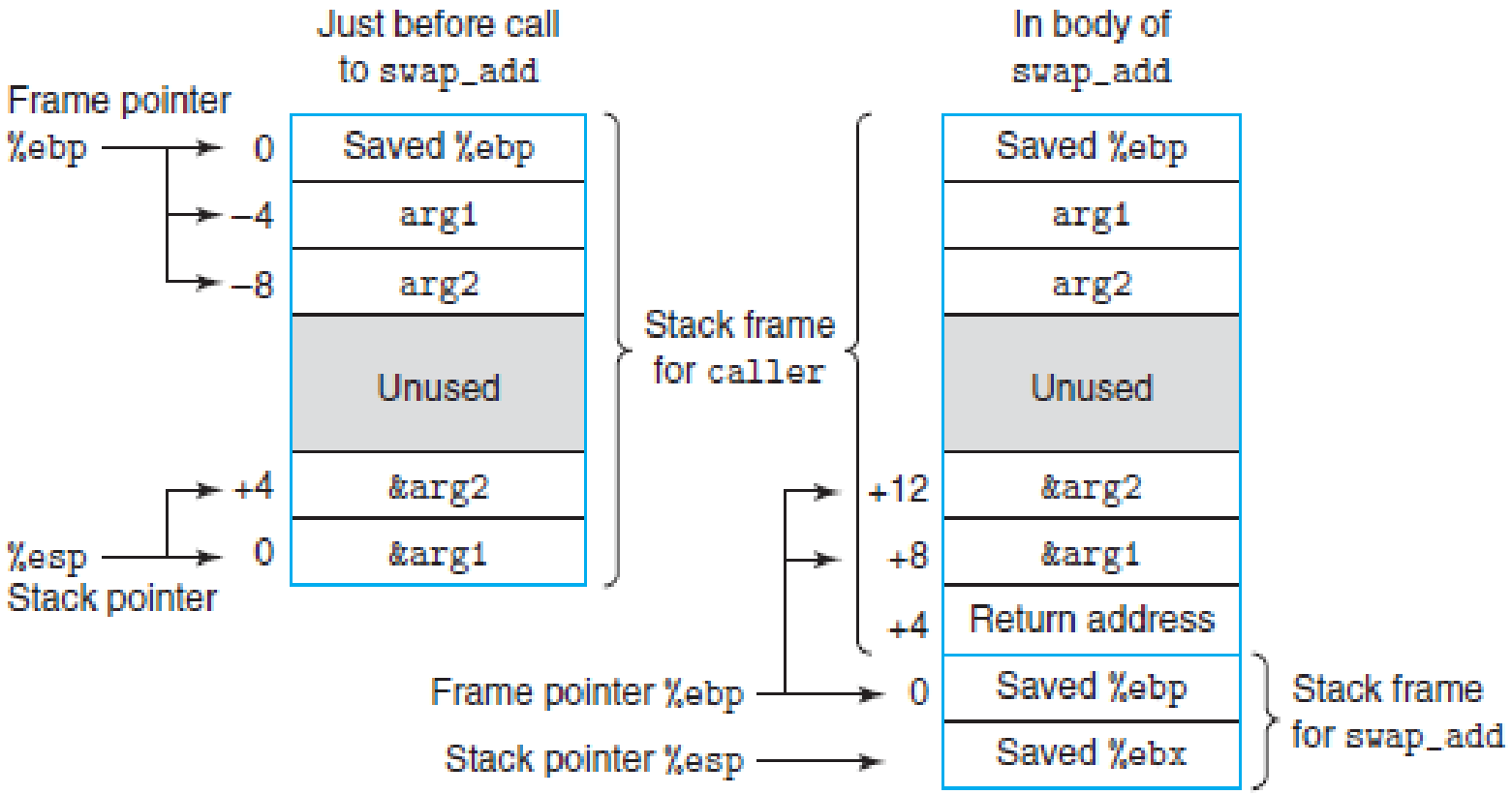


Figure 3.24 **Stack frames for `caller` and `swap_add`.** Procedure `swap_add` retrieves its arguments from the stack frame for `caller`.