Intro to C

- Functions
- Addresses and Memory Model
- Pointers
- Pointer Variables
- Pointers and Arrays

Functions

- function syntax:

```
[<storage class>] <return type>
   name (<parameters>) {
        <statements>
}
```

- parameter syntax:

  <type> varname , <type> varname> , …

- type **void**:
  - if **<return type>** is **void** the function has no return value
  - if **<parameters>** is **void** the function has no parameters
  - e.g., **void f(void);**

Functions

- example:

```
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    return 0;
}
```

- example:

```
double fmax(double x, double y) {
    if (x > y) {
        return x;
    } else {
        return y;
    }
}
```

Parameter passing

- C implements **call-by-value** parameter passing:

```
/* Formal parameters: m, n */

int maxint(int m, int n) {
   if (m > n) {
      return m;
   } else {
      return n;
   }
}
```

```
/* ... more code ... */

void some_function() {
    int a = 5;
    int b = 10;
    int c;

    /* Actual parameters: a, b */
    c = maxint (a, b);
    printf ("maximum of %d and %d is: %d", a, b, c);
}
```

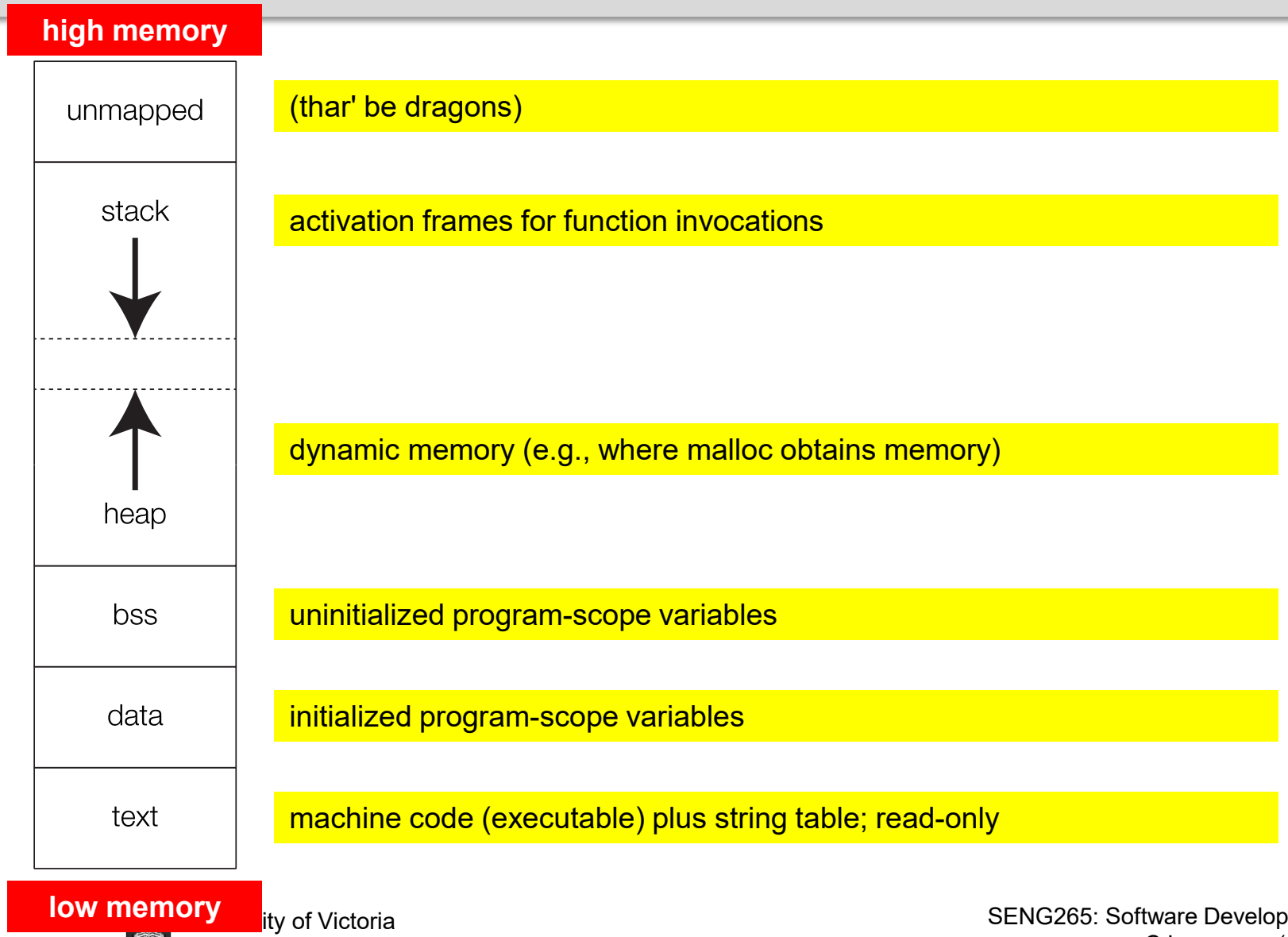
Parameter passing

- **Call-by-value semantics** copies actual parameters into formal parameters.

```
int power2( double f ) {
    if (f > sqrt(DBL_MAX)) {
        return 0;  /* Some sort of error was detected... */
    } else {
        return (int) (f * f);
    }
}
```

```
/* ... a bunch of code intervenes ... */

void some_other_function() {
    double g = 4.0;
    int h = power2(g);

    printf( "%f %d \n", g, h );
}
```

Memory model



Notation (declaration)

- **Pointer variables** are declared in terms of other types (scalar and nonscalar)
- More accurate to read the simpler variable declarations **right-to-left**

```
int *a;
double *f;
```

‘a’	is a variable	that holds an address	**to an ‘int’**
‘f’	is a variable	that holds an address	**to a ‘double’**

Notation (expression)

&x "get the address of memory location used to store the variable x" (**referencing**)

***y** "read y – which contains the address to some variable – and then go to that address in order to read what is there" (**dereferencing**)

- Remember that * can appear in a **variable declaration**
- **However, * has a different meaning in a declaration!**

```
double f = 30.0;
double *g = &f;
printf( "%lf  %lf\n", f, *g );
```

Addresses and Pointers

- Compare the following two code fragments

```
int x = 1;
int y = x;
x = 2;
printf("y is %d\n", y); /* “y is 1” */
```

```
int x = 1;
int *y = &x;
x = 2;
printf("*y is %d\n", *y); /* “*y is 2” */
```

- In other words, **x** is a synonym for ***&x**

Pointers

- Why do we need pointers?
- **Call-by-value** works well for passing parameters into functions, but:
  - What if we want values to be modified in the call function?
  - What if want to pass a large struct as a function argument?
- Functions can only return a single value in **return** statements.
  - What if we need multiple values changed (but don't want to write a struct for this)?
  - **Call-by-reference-like** semantics would get around the limitation of a "single return value".
  - **However, C only has call-by-value semantics!**
  - (C++ has call-by-value and call-by-reference)

Example

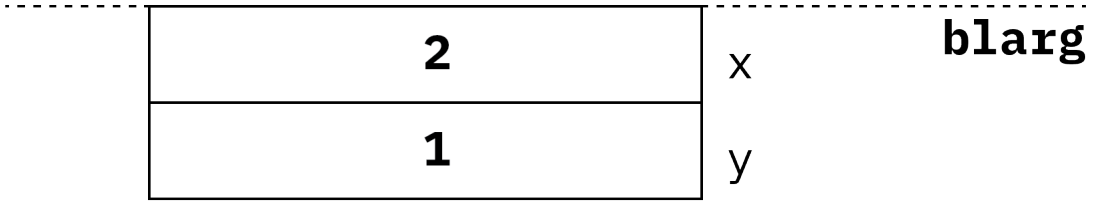
- swap function:

```
void swap(int a, int b) {
        int temp = a;
        a = b;
        b = temp;
}
```

```
/* ... some code here ... */

void blarg() {
    int x = 2;
    int y = 1;

    swap(x, y);
    printf("x = %d, y = %d\n", x, y); /* x = 2, y = 1 */
}
```

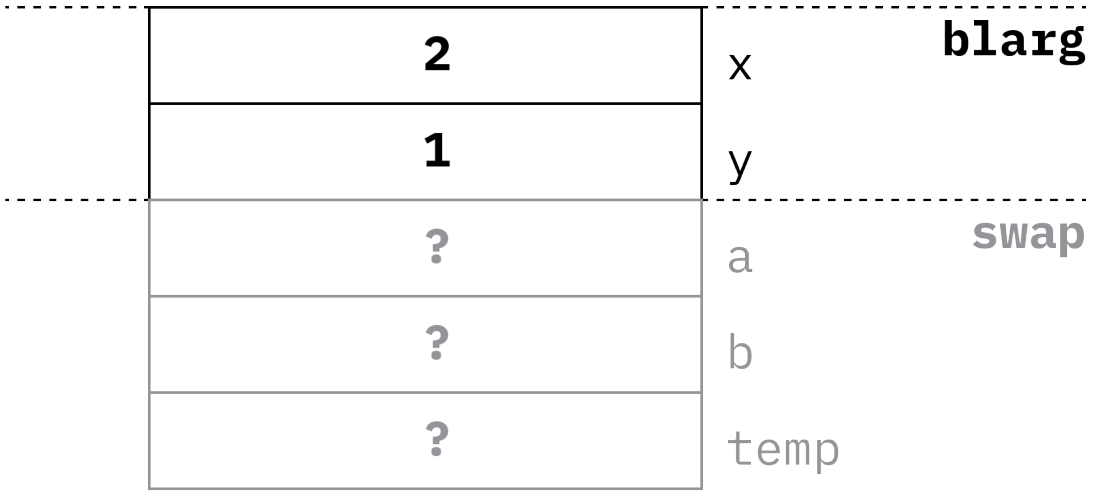
```
void swap(int a, int b) {
        int temp = a;
        a = b;
        b = temp;
}
```

```
/* ... some code here ... */

void blarg() {
    int x = 2;
    int y = 1;

    swap(x, y);
    printf("x = %d, y = %d\n", x, y); /* x = 2, y = 1 */
}
```



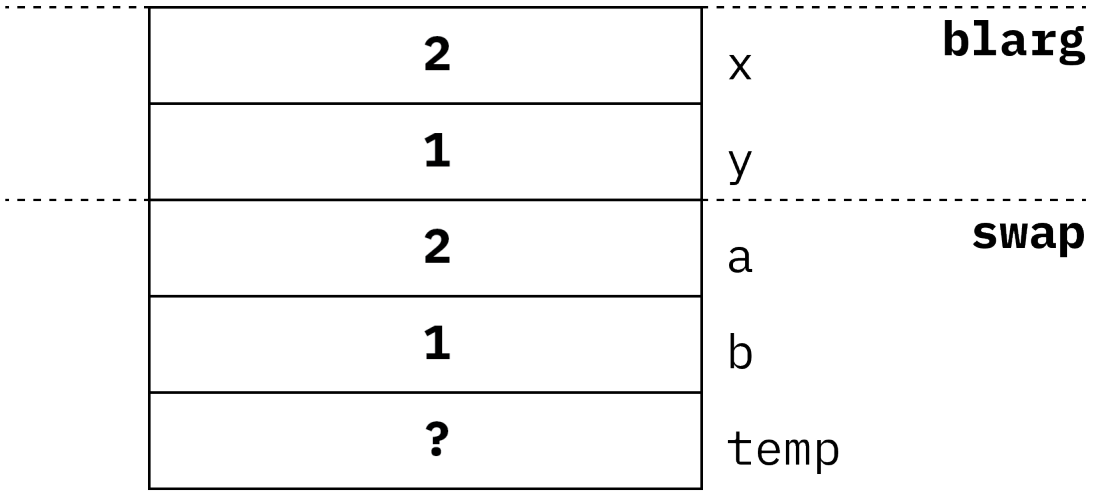
```
void swap(int a, int b) {
        int temp = a;
        a = b;
        b = temp;
}
```

```
/* ... some code here ... */

void blarg() {
    int x = 2;
    int y = 1;

    swap(x, y);
    printf("x = %d, y = %d\n", x, y); /* x = 2, y = 1 */
}
```



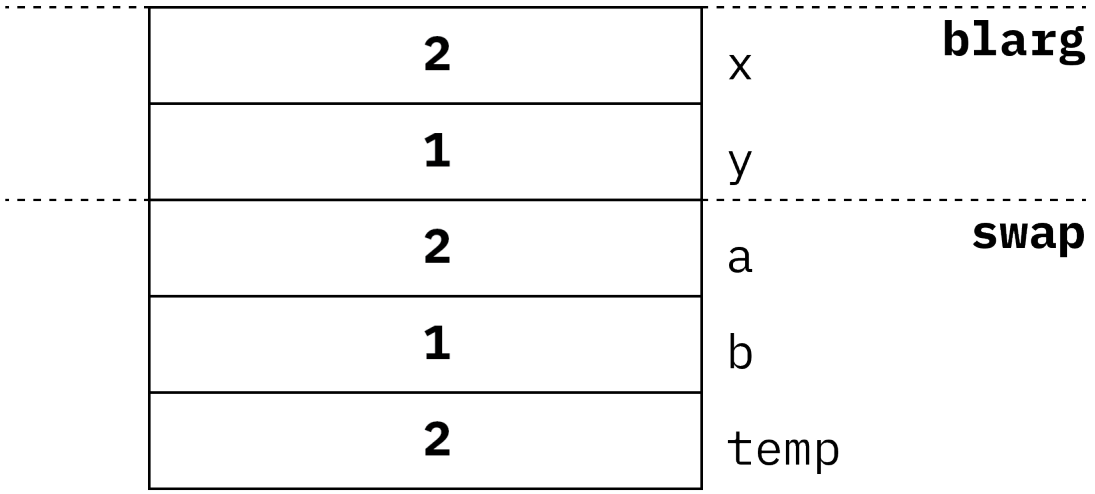
```
void swap(int a, int b) {
        int temp = a;
        a = b;
        b = temp;
}
```

```
/* ... some code here ... */

void blarg() {
    int x = 2;
    int y = 1;

    swap(x, y);
    printf("x = %d, y = %d\n", x, y); /* x = 2, y = 1 */
}
```



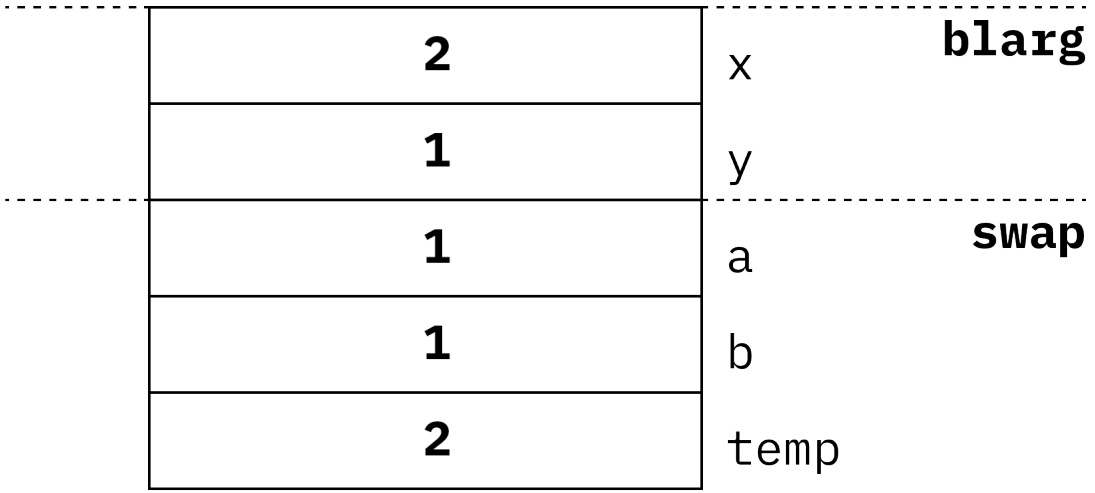
```
void swap(int a, int b) {
        int temp = a;
        a = b;
        b = temp;
}
```

```
/* ... some code here ... */

void blarg() {
    int x = 2;
    int y = 1;

    swap(x, y);
    printf("x = %d, y = %d\n", x, y); /* x = 2, y = 1 */
}
```



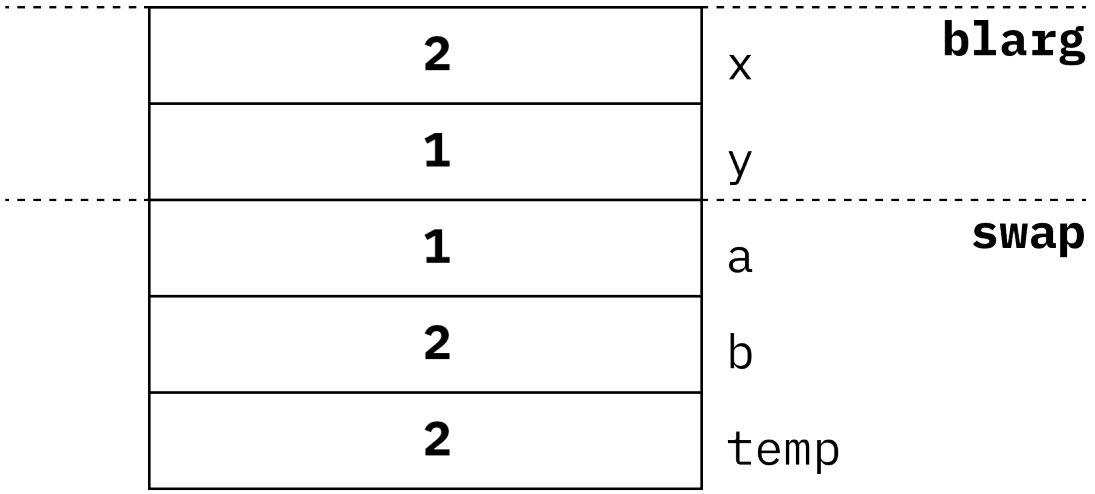
```
void swap(int a, int b) {
        int temp = a;
        a = b;
        b = temp;
}
```

```
/* ... some code here ... */

void blarg() {
    int x = 2;
    int y = 1;

    swap(x, y);
    printf("x = %d, y = %d\n", x, y); /* x = 2, y = 1 */
}
```



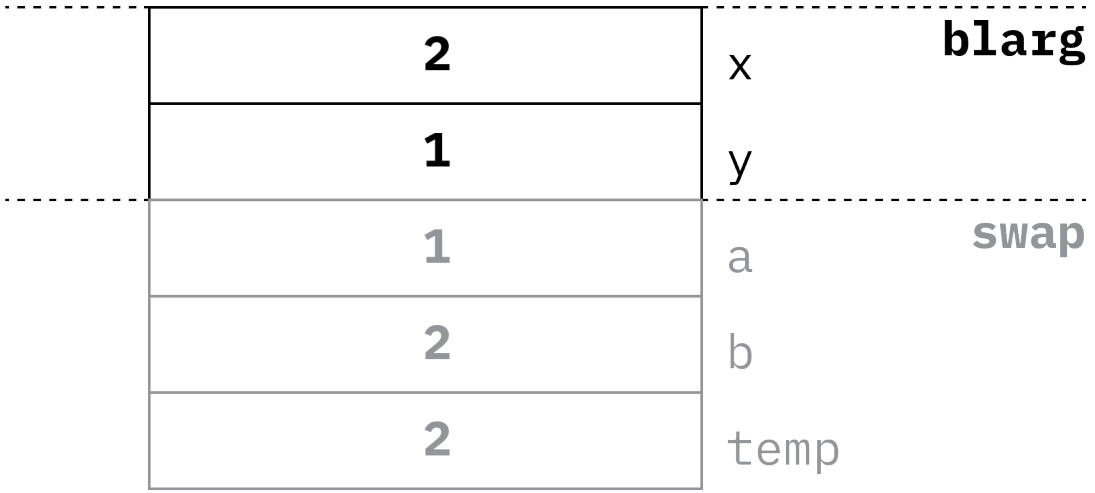
```
void swap(int a, int b) {
        int temp = a;
        a = b;
        b = temp;
}
```

```
/* ... some code here ... */

void blarg() {
    int x = 2;
    int y = 1;

    swap(x, y);
    printf("x = %d, y = %d\n", x, y); /* x = 2, y = 1 */
}
```



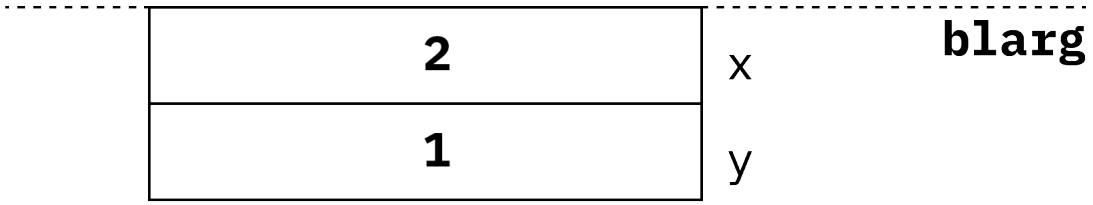
```
void swap(int a, int b) {
        int temp = a;
        a = b;
        b = temp;
}
```

```
/* ... some code here ... */

void blarg() {
    int x = 2;
    int y = 1;

    swap(x, y);
    printf("x = %d, y = %d\n", x, y); /* x = 2, y = 1 */
}
```



```
void swap(int a, int b) {
        int temp = a;
        a = b;
        b = temp;
}
```

```
/* ... some code here ... */

void blarg() {
    int x = 2;
    int y = 1;

    swap(x, y);
    printf("x = %d, y = %d\n", x, y); /* x = 2, y = 1 */
}
```



Example

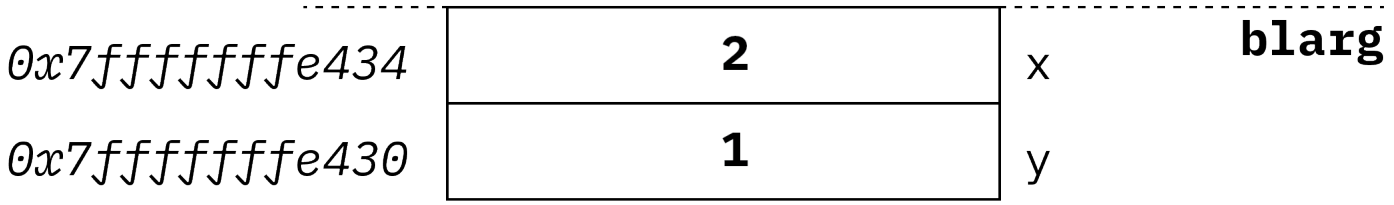
- swap function:

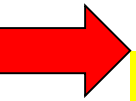
```
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```
/* ... some code here ... */

void blarg() {
    int x = 2;
    int y = 1;

    swap(&x, &y);
    printf("x = %d, y = %d\n", x, y);
}
```

```
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

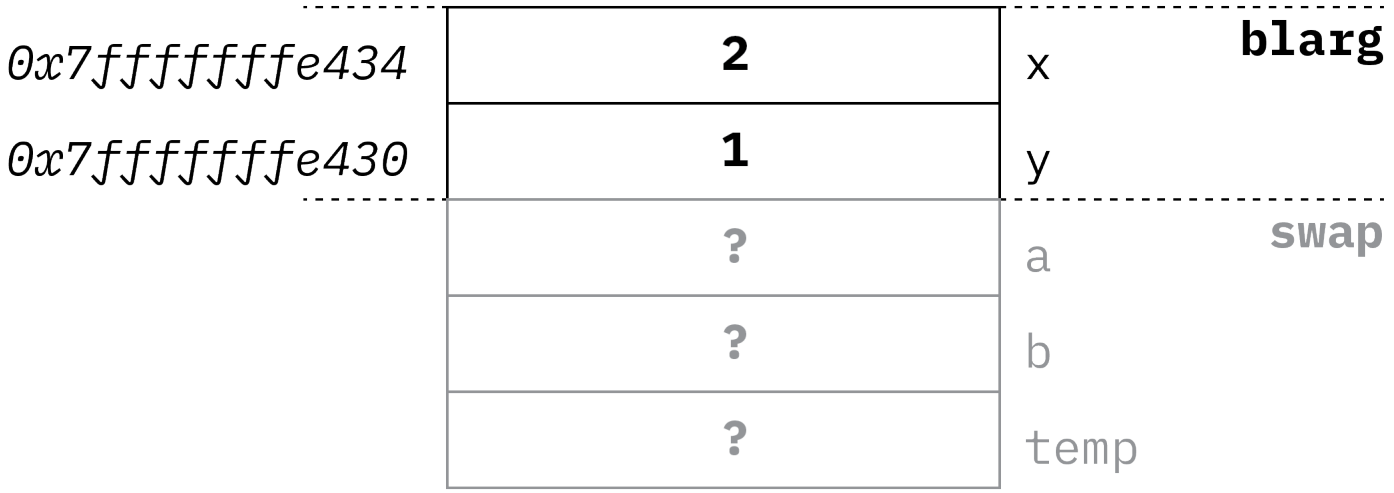
```
/* ... some code here ... */

void blarg() {
    int x = 2;
    int y = 1;

    swap(&x, &y);
    printf("x = %d, y = %d\n", x, y);
}
```

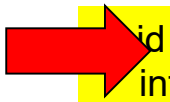




```
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

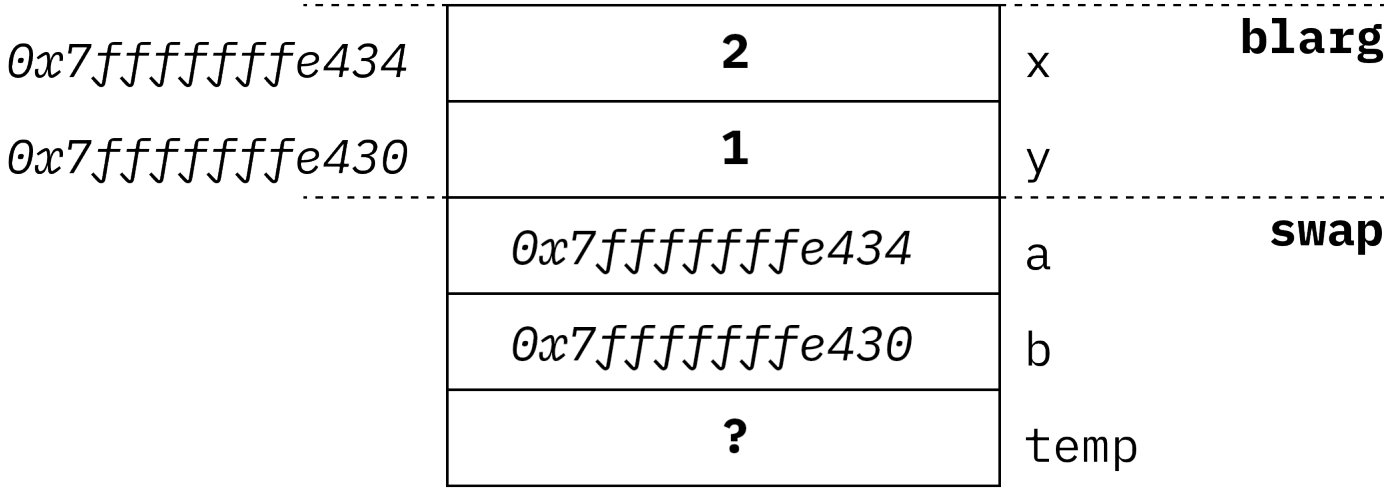
```
/* ... some code here ... */

void blarg() {
    int x = 2;
    int y = 1;

    swap(&x, &y);
    printf("x = %d, y = %d\n", x, y);
}
```





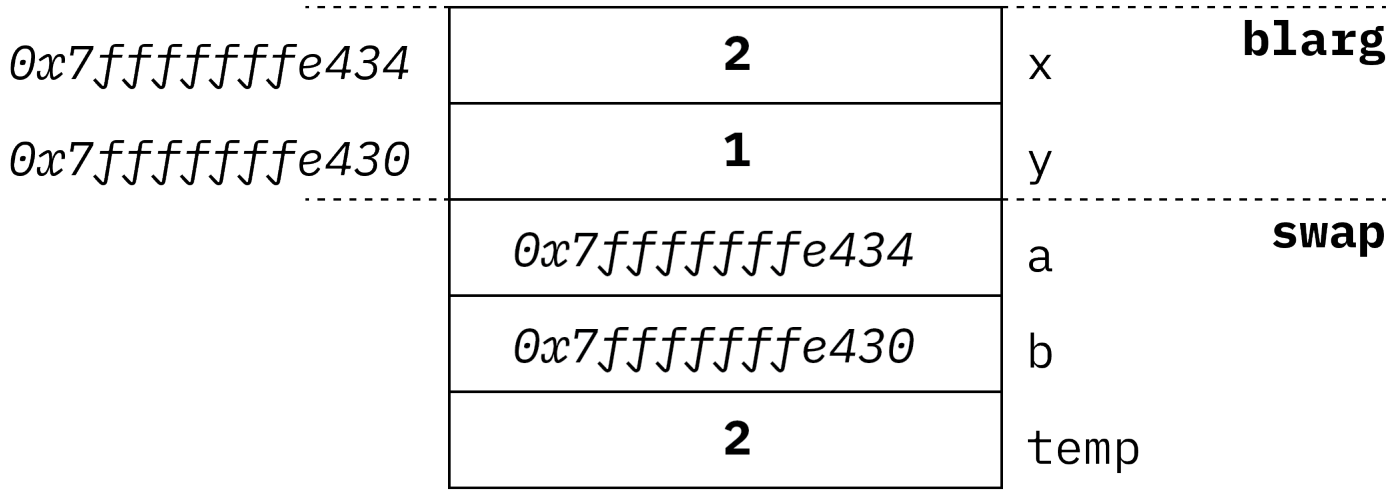
```
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```
/* ... some code here ... */

void blarg() {
    int x = 2;
    int y = 1;

    swap(&x, &y);
    printf("x = %d, y = %d\n", x, y);
}
```



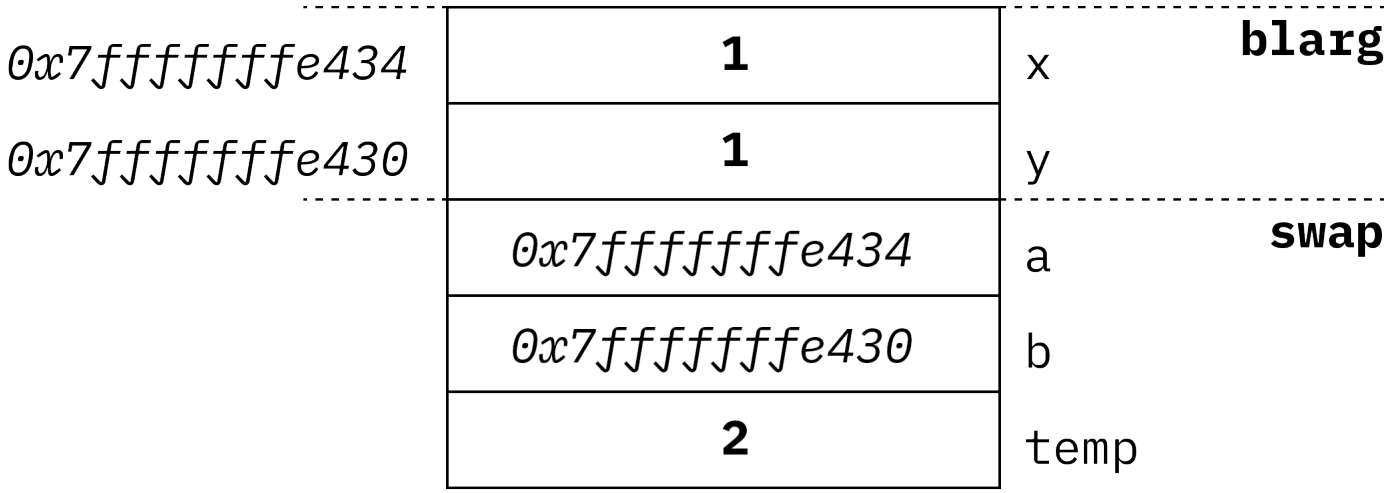
```
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```
/* ... some code here ... */

void blarg() {
    int x = 2;
    int y = 1;

    swap(&x, &y);
    printf("x = %d, y = %d\n", x, y);
}
```



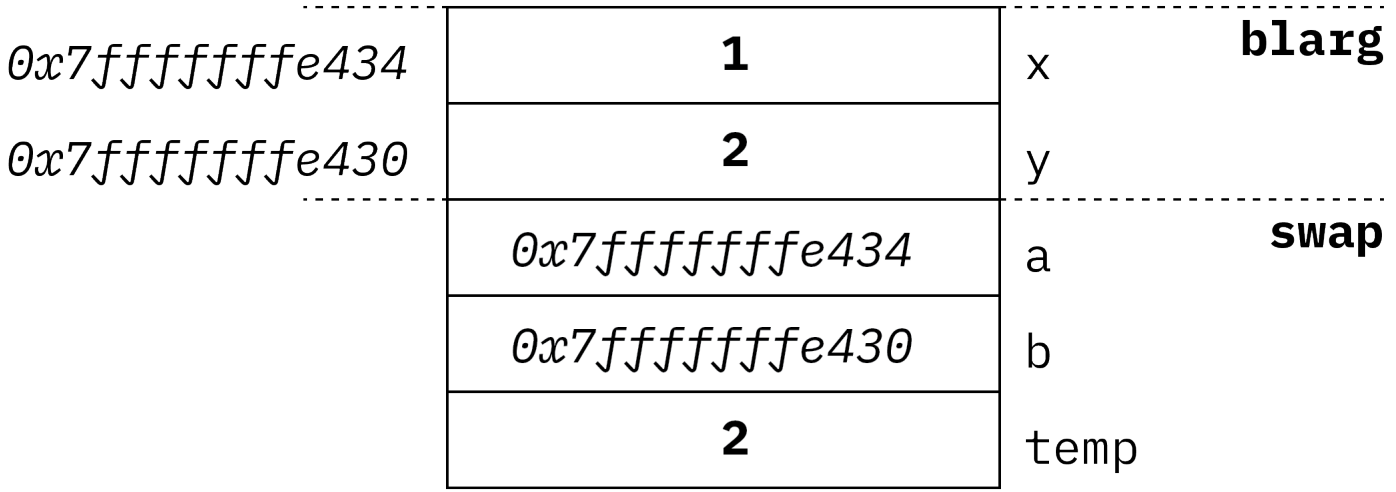
```
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```
/* ... some code here ... */

void blarg() {
    int x = 2;
    int y = 1;

    swap(&x, &y);
    printf("x = %d, y = %d\n", x, y);
}
```



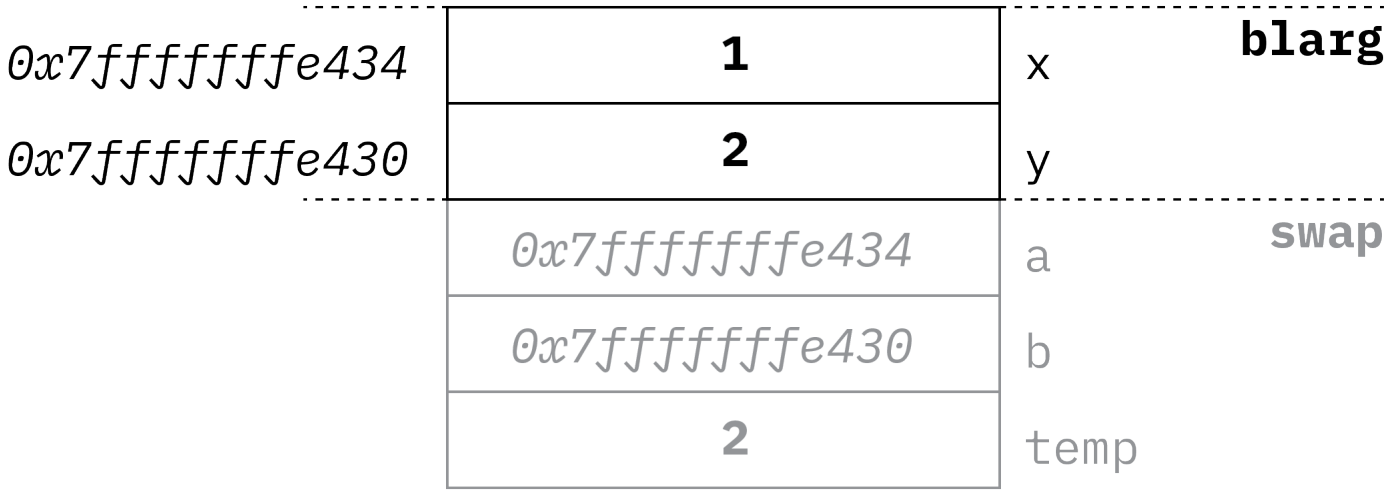
```
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```
/* ... some code here ... */

void blarg() {
    int x = 2;
    int y = 1;

    swap(&x, &y);
    printf("x = %d, y = %d\n", x, y);
}
```



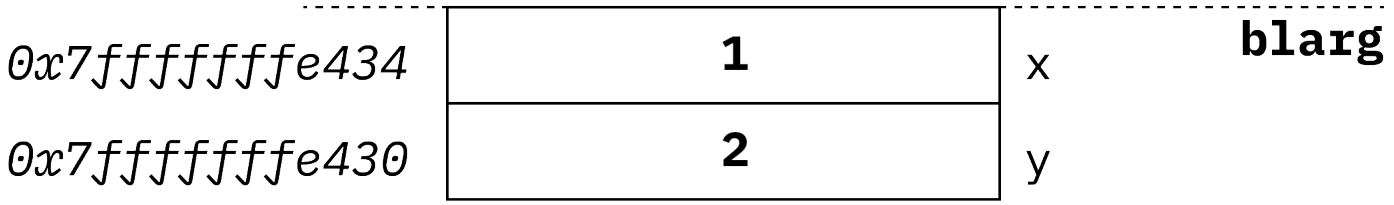
```
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```
/* ... some code here ... */

void blarg() {
    int x = 2;
    int y = 1;

    swap(&x, &y);
    printf("x = %d, y = %d\n", x, y);
}
```



```
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```
/* ... some code here ... */

void blarg() {
    int x = 2;
    int y = 1;

    swap(&x, &y);
    printf("x = %d, y = %d\n", x, y);
}
```



Pointers and arrays

- Recall that arrays are an aggregate data type where each data element has the same "type":

```
int grades[10];
struct date_record info[50];
char buffer[100];
```

- All elements in an array occupy contiguous memory locations
- To get the address of any data element, we can use **&**:

```
5th element of "grades": &grades[4]
1st element of info: &info[0]
last element of "buffer": &buffer[99]
```

Pointers and arrays

- An important array location is usually that of the first element
- In C, an array variable name without the subscript represents the first element; recall that each element is a character

```
char  buffer[100];
char *cursor;

cursor = &buffer[0]; /* these two lines ... */
cursor = buffer;     /* ... have the same effect. */
```

Pointers and arrays (3)

- Can use pointer variables and array names (sometimes) interchangeably to access array elements:

```
int X[4];
int *p = &X[0];
p = X; /* okay */
p++;   /* okay */
X = p; /* illegal */
X++;   /* illegal */
X[1] ~ *(p + 1);
X[n] ~ *(p + n);
```

- Declarations: the following function declarations are equivalent:

```
extern double func(double X[]);
extern double func(double *X);
```

  - Format #1 is often preferred as it does conveys more information