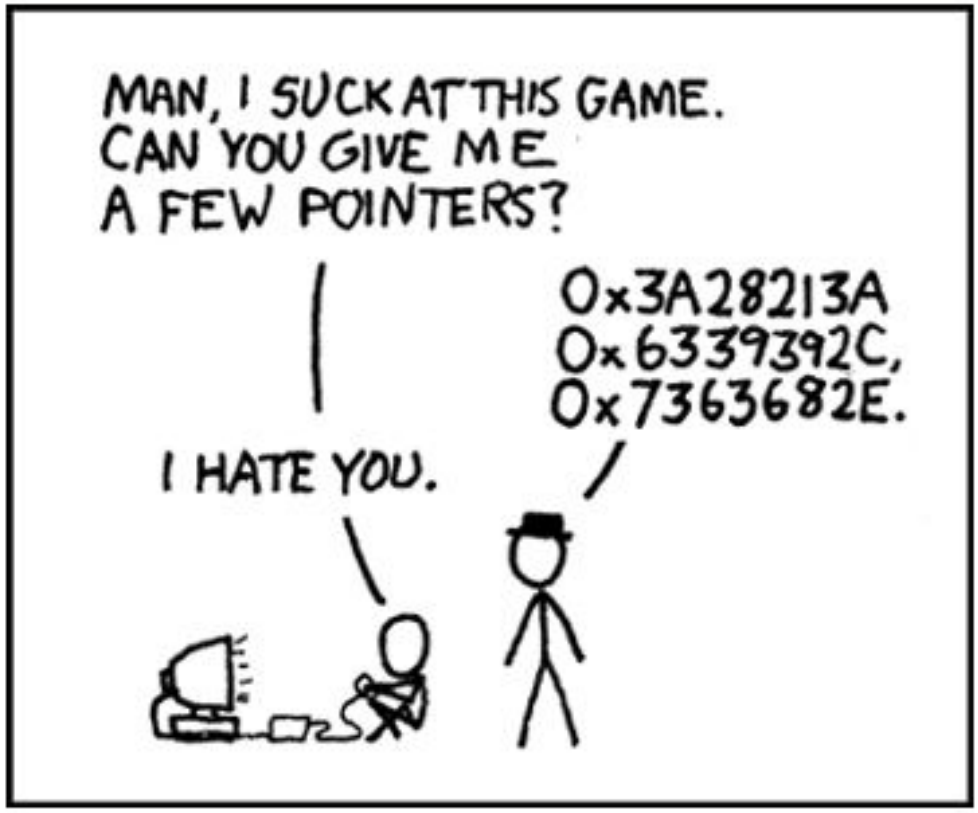
4. Pointers

The C language ... then taught two entire generations of programmers to ignore buffer overflows, and nearly every other exceptional condition, as well.

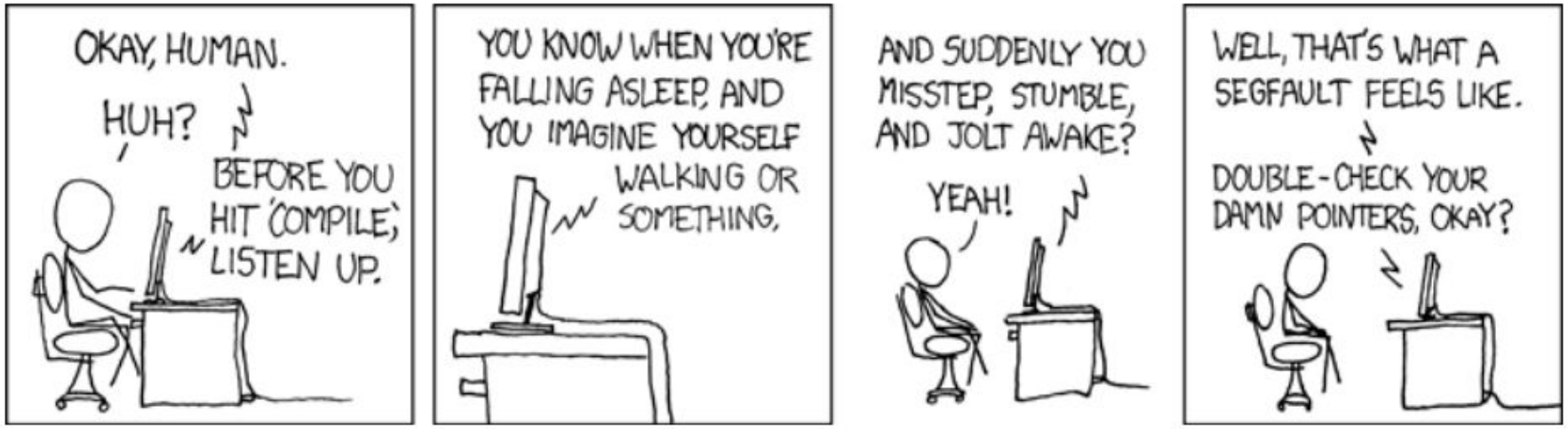
-- Henry Baker in "Buffer Overflow Security Problems".



Crash alert!!

- In C, all of memory is just one huge array
- There are no run-time checks
  - Except the memory protection provided by the OS
    - Stops you accessing device registers, other people's programs, etc
  - And the *stack smashing* check at the end of each function call
- Programming errors with pointers/arrays usually results in a crash!
  - "Segmentation fault, core dumped" [Linux]
  - "This program has encountered a problem and needs to close" [Windows]
- Sometimes a corrupted memory location causes a problem much later in the execution
  - The worst sort of problem to debug

xkcd on segmentation faults



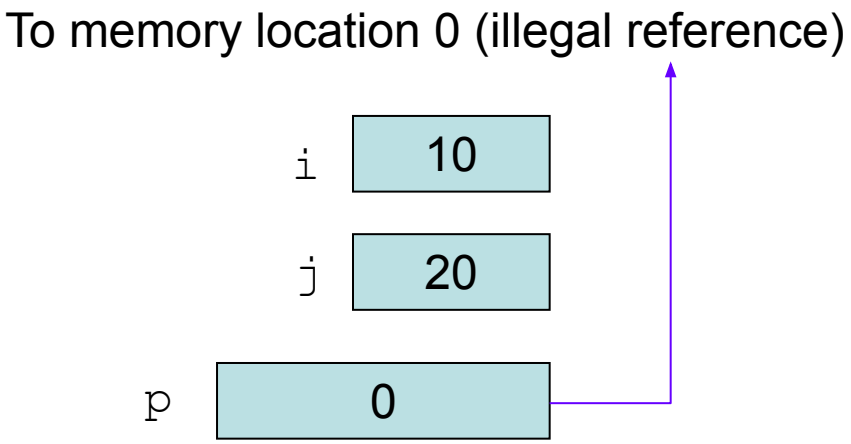
|< < PREV RANDOM NEXT > >|

PERMANENT LINK TO THIS COMIC: HTTP://XKCD.COM/371/

Pointers

- A pointer stores the address of another variable (or function)

```
int i = 10;
int j = 20;
int* p = NULL;
p = &i;
*p = j; // i = j
```

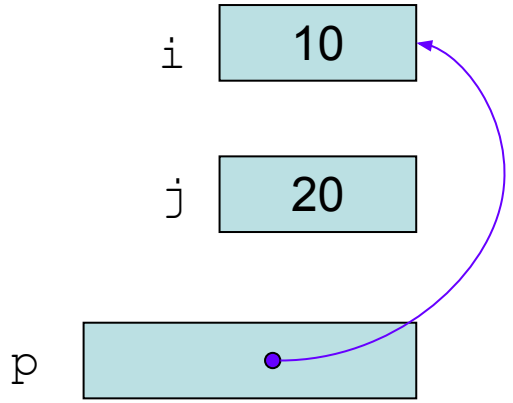


- The *declaration* `int* p = NULL` should be read as "p is a pointer to an int and the pointer is initialised to NULL (0)".
- The textbook writes pointer declarations as *int *p = NULL.*
  - This is equivalent; space is not significant here.
  - But textbook form is suggestive of an *assignment* `*p = NULL` which would at this stage cause a segmentation fault by trying to set location zero in memory to NULL!!

Pointers

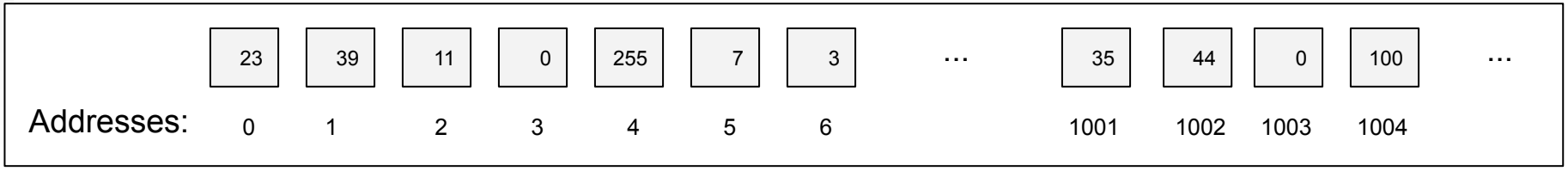
- A pointer stores the address of another variable (or function)

```
int i = 10;
int j = 20;
int* p = NULL;
p = &i;  // p = the address of i
*p = j;  // i = j
```



& means "the address of"

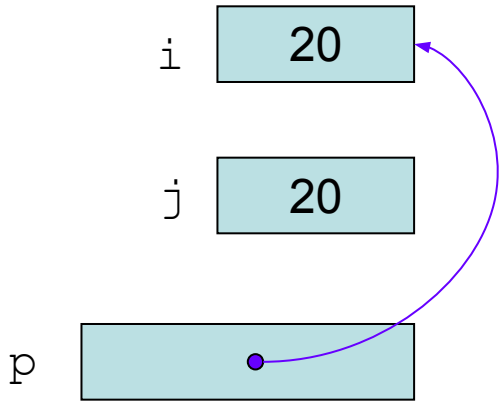
Remember:



Pointers

- A pointer stores the address of another variable (or function)

```
int i = 10;
int j = 20;
int* p = NULL;
p = &i;
*p = j;    // i = j
```



In **declarations**, read '*' as '*-pointer*', e.g. *int* p* declares an *int-pointer p*

In **expressions**, read '*' as *"indirectly via"* so **p = j* is *Get the value of j and store it indirectly via the pointer p*. Similarly, *j = *p* would be *Get a value indirectly via p and store it in j*.

Video

http://www.youtube.com/watch?v=6pmWojisM_E&feature=player_embedded

Uses of pointers

FTSE: "We can solve any problem by introducing an extra level of indirection"

1. As references to other variables (or functions)

   (a) Reference parameters

   – e.g. *scanf*; *swap* (next slide); *getTwoRandoms*

   (b) References to dynamically-allocated data structures

   – Linked lists, etc. See next two weeks' notes.

2. For fast and compact array/string manipulation. Except:

   – it's probably *not* significantly faster

   – "compact" => "unreadable"!

   • Not encouraged!

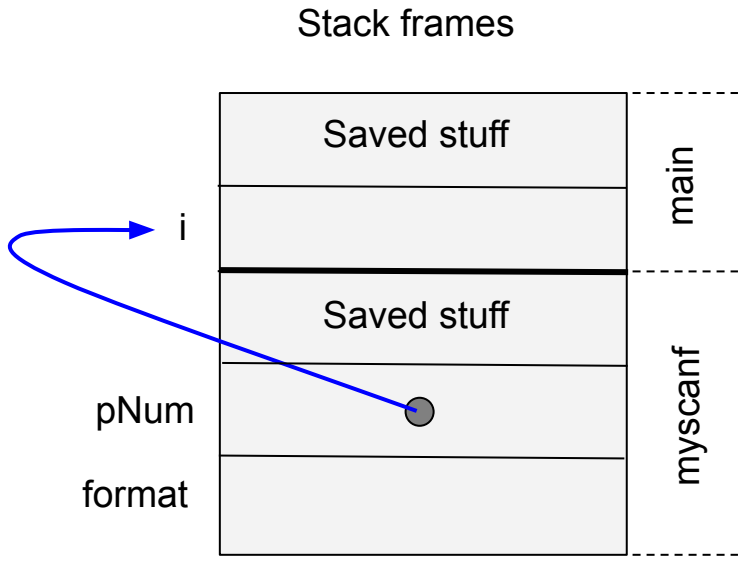
Pointers as references: myscanf

Assumes format is “%d”

```
#include <stdio.h>
#include <ctype.h>  // For isdigit

void myscanf(char format[], int* pNum)
{
    int num = 0;
    int c = 0;
    while (isdigit((c = getchar())) {
        num = num * 10 + (c - '0');
    }
    *pNum = num;
}

int main(void)
{
    int i = 0;
    myscanf("%d", &i);
    printf("i = %d\n", i);
}
```



Example 2: swap

```
// Swap the values pointed to by p1 and p2
void swap(int* p1, int* p2)
{
    ...  // You write it!
}

int main(void)
{
    int data1[ ] = { ....   /* numbers */  .... }
    int data2[ ] = { ....   /* more numbers */ ....  }
    int longer = sizeof(data1) / sizeof(int); // length of data1
    int shorter= sizeof(data2) / sizeof(int); // length of data2
    if (longer < shorter) {
        swap(&longer, &shorter);
    }

    // Now variable longer is the length of the longer of the two
    // arrays, and variable shorter is the length of the shorter.
```

The meaning of assignment

This may or may not help. If not, ignore it!

- Consider:

```
int i, j;
i = j;
```

- There's an asymmetry here:
  - RHS is an expression, in this case the *value* stored in *j*
  - LHS is the variable *i* itself, as a target
- To understand C properly, you should think of *i* and *j* as the *addresses* of the variables in memory
  - c.f., in assembly language: *movl j, i*
  - Means "move (i.e. copy) a long value from memory location *j* to memory location *i*"

Assignment (con'td)

- In C, assignment statements are of the form *expression* = *expression* but
  - The expression on the left of the assignment must be an *l-value* (= "LHS value"), i.e., it must be an address in memory.
- When *evaluating* a RHS expression, addresses are **automatically dereferenced**, i.e., the value at the address is used rather than the address itself.
- So given $i = j$, both sides are *l*-values, but the RHS is automatically dereferenced to yield a value.
- C's * and & operators respectively add and remove one level of indirection
  - Can think of '&' as suppressing the usual automatic dereferencing

Pointers and arrays

- An array name is *almost* exactly the same as a pointer to the zeroth element, except it's *const*

```
int x, a[5], b[5];
int* pa = NULL;
int* pb = NULL;
pa = a;         // Same as pa = &a[0]
x = *pa;        // Same as x = a[0], x = *a or x = pa[0]!
x = *(pa+2);    // Same as x = a[2], x = *(a+2), x = pa[2]
a += 1;         // Illegal - a is const
pa += 1;        // OK - steps to next element in array
b = a;          // Illegal. b is const. Use memcpy.
pb = pa;        // OK
for (pa = a; pa < &a[5]; pa++) {  // OK, but why not ...
    *pa = 0;                      // ... use subscripts?
}
```

WARNING! NOT: *int* pa=NULL, pb=NULL;*

C doesn't really have arrays

- It only has "subscripting" via pointer arithmetic

```
int b[10];
b[3] = 17;          // Just a shorthand for ...
*(b + 3) = 17;      // ... this!
3[b] = 17;          // So this is legal too!!!
```

- Remember: outside the scope of the declaration, there's no way of knowing how large an array is, so no way of checking for subscript or pointer out-of-range.
  - But obviously the program that declares the array knows its size
  - *sizeof (b)* is 40, c.f. *sizeof(char*)* which is 8 (on 64-bit machine).
    - But this doesn't work when *b* is a formal parameter

Array parameters

- Array name `a` passed as actual parameter is equivalent to passing `&a[0]`
- In formal parameter declarations, `char p[]` and `char* p` are equivalent. [This could be `int* data`]

```
int max(int n, int data[] )
{
    int big = data[0];
    int i = 1;
    for (; i < n; i++) {
        if (data[i] > big) {
            big = data[i];
        }
    }
    return big;
}
```

=

```
int max(int n, int* pData)
{
    int* endOfData = pData + n;
    int big = *pData++;
    for (; pData < endOfData; pData++) {
        if (*pData > big) {
            big = *pData ;
        }
    }
    return big;
}
```

```
int nums[ ] = {1, 4, 11, -19, 21, 99, 44};
printf("Maximum value of nums = %d\n", max(7, nums));
```

Style: use of const

- When passing arrays as parameters, use *const* in signature if function doesn't modify array.
  - You can't write to something that a *const* pointer points at
    - You can still modify the *pointer* though!
- Stops inadvertent modifying of array
- Makes it clear to user that array is for input only
- Prevents wrong calls when have one input array and one output array

```
int max(int n, const int* pData)
{
    const int* endOfData = pData + n;
    int big = *pData++;
    for (; pData < endOfData; pData++) {
        if (*pData > big) {
            big = *pData ;
        }
     }
     return big;
}
```