Pointers in C

C Programming and Software Tools

N.C. State Department of Computer Science
If ever there was a time to pay attention, now is that time.
A critical juncture

When you understand pointers

If you don’t…
Agenda

• I’m going to cover this TWICE, in two different ways

  – My condensed slides

  – The original slides
Pointers: the short, short version
Memory is a real thing!

- Most languages – protected variables
- C – flat memory space

Figure from Rudra Dutta, NCSU, 2007
The memory map on 32-bit x86

- Static data
- Heap
- Shared library
- Kernel space

- Func params
- Bookkeeping (frame & stack pointers)
- Local variables

Frame 0
Frame 1
Frame 2+

Based on Dawn Song’s RISE: http://research.microsoft.com/projects/SWSecInstitute/slides/Song.ppt
What do variable declarations do?

```c
int x = 5;
char msg[] = "Hello";

int main(int argc, const char* argv[]) {
    int v;
    float pi = 3.14159;

    printf("%d\n", x);
    printf("%d\n", v);
}
```

When the program starts, set aside an extra 4 bytes of static data, and set them to 0x00000005. When I type x later, assume I want the value stored at the address you gave me.


Whenever this function is run, reserve a chunk of space on the stack. Put in it what was passed in; call it argc and argv.

In that chunk of stack space, reserve 4 more bytes. Don’t pre-fill them. When I type v later, give me the data in the spot chosen.

Ditto, but treat the space as a decimal, call it pi, and make it 3.14159.

Look up what’s in x and print it. Ditto for v.
What do variable declarations do?

```c
int x = 5;
char msg[] = "Hello";

int main(int argc, const char* argv[]) {
    int v;
    float pi = 3.14159;

    printf("%d\n", x);
    printf("%d\n", v);
}
```
Let’s look at memory addresses!

• You can find the address of ANY variable with:

The address-of operator

```
int v = 5;
printf("%d\n", v);
printf("%p\n", &v);
```

$ gcc x4.c && ./a.out
5
0x7fffd232228c
Testing our memory map

```c
int x=5;
char msg[] = "Hello";

int main(int argc, const char* argv[]) {
    int v;
    float pi = 3.14159;

    printf("&x: %p\n", &x);
    printf("&msg: %p\n", &msg);
    printf("&argc: %p\n", &argc);
    printf("&argv: %p\n", &argv);
    printf("&v: %p\n", &v);
    printf("&pi: %p\n", &pi);
}
```

$ gcc x.c && ./a.out
&x: 0x601020
&msg: 0x601024
&argc: 0x7fff85b78c2c
&argv: 0x7fff85b78c20
&v: 0x7fff85b78c38
&pi: 0x7fff85b78c3c
What’s a pointer?

- It’s a **memory address** you treat as a **variable**
- You declare pointers with:

```
int v = 5;
int* p = &v;
printf("%d\n", v);
printf("%p\n", p);
```

The *dereference* operator

```
$ gcc x4.c && ./a.out
5
0x7fffe0e60b7c
```
What’s a pointer?

• You can **look up** what’s stored at a pointer!
• You **dereference** pointers with:

```
int v = 5;
int* p = &v;
printf("%d\n",v);
printf("%p\n",p);
printf("%d\n",*p);
```

The **dereference** operator

```
$ gcc x4.c && ./a.out
5
0x7fffe0e60b7c
5
```
What is an array?

• The shocking truth: You’ve been using pointers all along!

• Every array is a pointer to a block of memory

```c
int x = 9;
char msg[] = “hello”;
short nums = {6, 7, 8};
```
int x[] = {15,16,17,18,19,20};

<table>
<thead>
<tr>
<th>Array lookup</th>
<th>Pointer reference</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>int*</td>
</tr>
<tr>
<td>x[0]</td>
<td>*x</td>
<td>int</td>
</tr>
<tr>
<td>x[5]</td>
<td>*(x+5)</td>
<td>int</td>
</tr>
<tr>
<td>x[n]</td>
<td>*(x+n)</td>
<td>int</td>
</tr>
<tr>
<td>&amp;x[0]</td>
<td>x</td>
<td>int*</td>
</tr>
<tr>
<td>&amp;x[5]</td>
<td>x+5</td>
<td>int*</td>
</tr>
<tr>
<td>&amp;x[n]</td>
<td>x+n</td>
<td>int*</td>
</tr>
</tbody>
</table>

(In case you don’t believe me)

```
int n=2;
printf("%p %p\n", x , x);
printf("%d %d\n", x[0], *x);
printf("%d %d\n", x[5], *(x+5));
printf("%d %d\n", x[n], *(x+n));
printf("%p %p\n", &x[0], x);
printf("%p %p\n", &x[5], x+5);
printf("%p %p\n", &x[n], x+n);
```

$ gcc x5.c && ./a.out
0x7fffa2d0b9d0 0x7fffa2d0b9d0
15 15
20 20
17 17
0x7fffa2d0b9d0 0x7fffa2d0b9d0
0x7fffa2d0b9e4 0x7fffa2d0b9e4
0x7fffa2d0b9d8 0x7fffa2d0b9d8

• This is why arrays don’t know their own length: they’re just blocks of memory with a pointer!
Using pointers

• Start with an address of something that exists
• Manipulate according to known rules
• Don’t go out of bounds (don’t screw up)

```c
void underscorify(char* s) {
    char* p = s;
    while (*p != 0) {
        if (*p == ' ') {
            *p = '_';
        }
        p++;
    }
}

int main() {
    char msg[] = "Here are words";
    puts(msg);
    underscorify(msg);
    puts(msg);
}

$ gcc x3.c && ./a.out
Here are words
Here_are_words
```
void underscorify(char* s) {
    char* p = s;
    while (*p != 0) {
        if (*p == ' ') {
            *p = '_';
        }
        p++;
    }
}

// how a developer might code it
void underscorify2(char* s) {
    char* p;
    for (p = s; *p ; p++) {
        if (*p == ' ') {
            *p = '_';
        }
    }
}

// how a kernel hacker might code it
void underscorify3(char* s) {
    for ( ; *s ; s++) {
        if (*s == ' ') *s = '_';
    }
}
What happens if we run this?

```c
#include <stdio.h>

int main(int argc, const char* argv[]) {
    int* p;

    printf(" p:  %p\n", p);
    printf("*p:  %d\n", *p);
}
```

```bash
$ gcc x2.c && ./a.out
p:  (nil)
Segmentation fault (core dumped)
```
Pointers: powerful, but deadly

- Okay, I can fix this! I’ll initialize p!

```c
#include <stdio.h>

int main(int argc, const char* argv[]) {
    int* p = 100000;

    printf(" p:  %p\n",p);
    printf("*p:  %d\n",*p);
}
```

```
$ gcc x2.c
x2.c: In function ‘main’:
x2.c:4:9: warning: initialization makes pointer from integer without a cast [enabled by default]
$ .a.out
  p:  0x186a0
Segmentation fault (core dumped)
```
A more likely pointer bug...

```c
void underscorify_bad(char* s) {
    char* p = s;
    while (*p != '0') {
        if (*p == 0) {
            *p = '_';
        }
        p++;
    }
}

int main() {
    char msg[] = "Here are words";
    puts(msg);
    underscorify_bad(msg);
    puts(msg);
}
```
void underscorify_bad2(char* s) {
    char* p = s;
    while (*p != '0') {
        if (*p == ' ') {
            *p = '_';
        }
        p++;
    }
}

int main() {
    char msg[] = "Here are words";
    puts(msg);
    underscorify_bad2(msg);
    puts(msg);
}

Worked but crashed on exit
Worked totally!!
Worked totally!!
Worked totally!!
Worked totally!!
Worked totally!!
Worked totally!!
Worked totally!!
Effects of pointer mistakes

Access an array out of bounds or some other invalid pointer location?

- No visible effect
- Totally weird behavior
- Silent corruption & bad results
- Program crash with OS error

Example outputs:
- Segmentation fault (core dumped)
- Access violation at 0x00736002 (tried to read from 0x00000001F), program terminated.

Other examples:
- Mac OS X and other applications are not affected.
- Click Relaunch to launch the application again.
• **Memory is linear**, all the variables live at an address
  – Variable declarations reserve a range of memory space

• You can get the address of any variable with the **address-of operator** &
  ```c
  int x;  printf(“%p\n”, &x);
  ```

• You can **declare a pointer** with the **dereference operator** * appended to a type:
  ```c
  int* p = &x;
  ```

• You can find the data at a memory address with the **dereference operator** * prepended to a pointer expression:
  ```c
  printf(“%d\n”, *p);
  ```

• Arrays in C are just pointers to a chunk of memory

• Don’t screw up
POINTERS – TRADITIONAL SLIDES
The Derived Data Types

✓ Arrays

➢ Pointers
  • (Structs)
  • ( Enums)
  • ( Unions)
Pointers Every Day

• Examples
  – telephone numbers
  – web pages

• Principle: indirection

• Benefits?
All References are Addresses?

• In reality, all program references (to variables, functions, system calls, interrupts, ...) are addresses

  1. you write code that uses symbolic names
  2. the compiler translates those for you into the addresses needed by the computer
     – requires a directory or symbol table (name → address translation)

• You could just write code that uses addresses (no symbolic names)
  – advantages? disadvantages?
Pointer Operations in C

• Make sense?
  • "v and w are variables of type int"
  • "pv is a variable containing the address of another variable"
  • "pv = the address of v"
  • “v = the value of the int whose address is contained in pv"

```
int v, w;
int * pv;
pv = &v;
w = *pv;
```
C Pointer Operators

<table>
<thead>
<tr>
<th>px = &amp;x;</th>
<th>“px is assigned the address of x”</th>
</tr>
</thead>
<tbody>
<tr>
<td>y = *px;</td>
<td>“y is assigned the value at the address indicated (pointed to) by px”</td>
</tr>
</tbody>
</table>

- **px** is not an alias (another **name**) for the variable **x**; it is a variable storing the **location** (address) of the variable **x**
...Operators (cont’d)

\& = “the address of...”

“ap is a pointer to an int”

```c
int a;
int *ap;
ap = &a;
```

“ap gets the address of variable a”

“cp is a pointer to a char”

```c
char c;
char *cp;
cp = &c;
```

“cp gets the address of variable c”

“fp is a pointer to a float”

```c
float f;
float *fp;
fp = &f;
```

“fp gets the address of variable f”
...Operators (cont’d)

* = “pointer to...”

```
*ap = 33;
b = *ap;
```

“the variable ap points to (i.e., a) is assigned value 33”
“b is assigned the value of the variable pointed to by ap (i.e., a)”

```
*cp = 'Q';
d = *cp;
```

“the variable cp points to (i.e., c) is assigned the value ‘Q’”
“d is assigned the value of the variable pointed to by cp (i.e., c)”

```
*fp = 3.14;
g = *fp;
```

“the variable fp points to (i.e., f) is assigned value 3.14”
“g is assigned the value of the variable pointed to by fp (i.e., f)”
Side note: where to put the *

• How I write and think about pointers:
  - `int* x; // x is an int pointer`

• How many C programmers do:
  - `int *x; // x is a pointer, its type is int`

• What does this mean?
  - `int *x,y;`

    Equivalent to:

    - `int *x; // x is a pointer, its type is int`
      `int y; // ...and y is an int`
Variable Names Refer to Memory

• A C expression, **without pointers**

```
        a = b + c; /* all of type int */
```

**Symbol Table**

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>b</td>
</tr>
<tr>
<td>4</td>
<td>c</td>
</tr>
<tr>
<td>8</td>
<td>a</td>
</tr>
</tbody>
</table>

**“Pseudo-Assembler” code**

```
load int at address 0 into reg1
load int at address 4 into reg2
add reg1 to reg2
store reg2 into address 8
```
## Variables Stored in Memory

Almost all machines are **byte-addressable**, i.e., every byte of memory has a unique address.

<table>
<thead>
<tr>
<th>Addr</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Value of b</td>
</tr>
<tr>
<td>4</td>
<td>Value of c</td>
</tr>
<tr>
<td>8</td>
<td>Value of a</td>
</tr>
</tbody>
</table>

32 bits (4 bytes) wide
Pointers Refer to Memory Also

• A C expression, with pointers

```c
int *ap;
ap = &a;
*ap = b + c; /* all of type int */
```

Symbol Table

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>b</td>
</tr>
<tr>
<td>4</td>
<td>c</td>
</tr>
<tr>
<td>8</td>
<td>a</td>
</tr>
<tr>
<td>12</td>
<td>ap</td>
</tr>
</tbody>
</table>

“Pseudo-assembler” code

```plaintext
load address 8 into reg3
load int at address 0 into reg1
load int at address 4 into reg2
add reg1 to reg2
store reg2 into address pointed to by reg3
```
Pointers Refer... (cont’d)

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
<th>Variable Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Value of b</td>
<td>b</td>
</tr>
<tr>
<td>4</td>
<td>Value of c</td>
<td>c</td>
</tr>
<tr>
<td>8</td>
<td>Value of a</td>
<td>a</td>
</tr>
<tr>
<td>12</td>
<td>8 (address of a)</td>
<td>ap</td>
</tr>
</tbody>
</table>

32 bits (4 bytes) wide
Addresses vs. Values

```c
int a = 35;
int *ap;
ap = &a;
printf(“ a=%d\n &a=%u\n ap=%u\n *p=%d\n”,
       a,
       (unsigned int) &a,
       (unsigned int) ap,
       *ap);
```

• Result of execution

```
a = 35
&a = 3221224568
ap = 3221224568
*ap = 35
```
### C expression

```
char * ap = &a;
char ** app = &ap;
char *** appp = &app;
***appp = b + c;
```

<table>
<thead>
<tr>
<th>Var</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>8</td>
</tr>
<tr>
<td>ap</td>
<td>12</td>
</tr>
<tr>
<td>app</td>
<td>20</td>
</tr>
<tr>
<td>appp</td>
<td>16</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>4</td>
</tr>
</tbody>
</table>

### Contents

<table>
<thead>
<tr>
<th>Addr</th>
<th>Contents</th>
<th>Var</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Value of b</td>
<td>b</td>
</tr>
<tr>
<td>4</td>
<td>Value of c</td>
<td>c</td>
</tr>
<tr>
<td>8</td>
<td>Value of a</td>
<td>a</td>
</tr>
<tr>
<td>12</td>
<td>8 (addr of a)</td>
<td>ap</td>
</tr>
<tr>
<td>16</td>
<td>20 (addr of app)</td>
<td>appp</td>
</tr>
<tr>
<td>20</td>
<td>12 (addr of ap)</td>
<td>app</td>
</tr>
</tbody>
</table>

32 bits (4 bytes) wide
Flow of Control in C Programs

• When you call a function, how do you know where to return to when exiting the called function?
  – The call function information is pushed on the stack
  – The callee is processed
  – The last part of the callee (before popping from the stack) is the address of the caller (a pointer to the caller in memory)
  – Return value is a pointer to where value is stored in memory
Why Pointers?

• Indirection provides a level of flexibility that is immensely useful
  – “There is no problem in computer science that cannot be solved by an extra level of indirection.”

• Even Java has pointers; you just can’t modify them
  – e.g., objects are passed to methods by reference, and can be modified by the method
...Types (cont’d)

- Make sure pointer type agrees with the type of the operand it points to

```c
int i, *ip;
float f, *fp;

fp = &f;    /* makes sense */
fp = &i;    /* definitely fishy */
/* but only a warning */
```

Ex.: if you're told the office of an instructor is a mailbox number, that's probably a mistake
Pointer Type Conversions

- **Pointer casts are possible, but rarely useful**
  - Unless you’re creative and believe in yourself

```c
char * cp = ...;
float * fp = ...;
...
fp = (float *) cp; /* casts a pointer to a char * to a pointer to a float???
*/
```

Analogy: like saying a phone number is really an email address -- doesn’t make sense!
Fast inverse square root
One of the wonders of the modern age

• Why does this work?
  – Crazy math and/or magic
  – Read wikipedia for more info...

Actual source code from Quake III Arena

```c
float Q_rsqrt( float number )
{
    long i;
    float x2, y;
    const float threethirds = 1.5F;

    x2 = number * 0.5F;
    y = number;
    i = * ( long * ) &y;         // evil floating point bit level hacking
    i = 0x5f3759df - ( i >> 1 ); // what the fuck?
    y = * ( float * ) &i;
    y = y * ( threethirds - ( x2 * y * y ) );  // 1st iteration
    // y = y * ( threethirds - ( x2 * y * y ) ); // 2nd iteration, this can be removed
    return y;
}
```

Didn’t actually invent this, but people assume he did.
...Conversions (cont’d)

However, casts (implicit or explicit) of variables pointed to are useful

```c
float f;
int i;
char * ip = &i ;
...
f = * ip; /* converts an int to a float */
f = i ; /* no different! */
```
Find the Pointer Bloopers

Do any of the following cause problems, and if so, what type?

1. `ap = &c;`  
   - incompatible types

2. `*ap = 3333;`  
   - OK

3. `c = ap;`  
   - incompatible types

4. `c = *ap;`  
   - Overflow

```c
int a, b, *ap, *bp;
char c, d, *cp, *dp;
float f, g, *fp, *gp;
```
Bloopers (cont’d)

```c
int a, b, *ap, *bp;
char c, d, *cp, *dp;
float f, g, *fp, *gp;
```

5. `dp = ap;`  incompatible types

6. `dp = 'Q';` almost certainly a mistake

7. `fp = 3.14159;` forgot the *

8. `gp = &fp;` incompatible types

9. `*gp = 3.14159;` OK
... Bloopers (cont’d)

```c
int a, b, *ap, *bp;
char c, d, *cp, *dp;
float f, g, *fp, *gp;
```

10. `*fp = &gp;`
   - incompatible types

11. `&gp = &fp;`
   - & cannot be on left-hand-side of assignment

12. `b = *a;`
   - a is not a pointer

13. `b = &a;`
   - b is not a pointer
### Ethical, cool things to do

- **OK:**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a = *p2;</code></td>
<td>copy value pointed to by <code>p2</code> to <code>a</code></td>
</tr>
<tr>
<td><code>*p1 = 35;</code></td>
<td>set value of variable pointed to by <code>p1</code> to 35</td>
</tr>
<tr>
<td><code>*p1 = b;</code></td>
<td>copy value of <code>b</code> to value pointed to by <code>p1</code></td>
</tr>
<tr>
<td><code>*p1 = *p2;</code></td>
<td>copy value pointed to by <code>p2</code> to value pointed to by <code>p1</code></td>
</tr>
<tr>
<td><code>p1 = &amp; b;</code></td>
<td><code>p1</code> gets the address of <code>b</code></td>
</tr>
<tr>
<td><code>p1 = p2;</code></td>
<td><code>p1</code> gets the address stored in <code>p2</code> (i.e., they now point to the same location)</td>
</tr>
</tbody>
</table>

Initially:

```c
int a, b, *p1, *p2;
a = 30, b = 50;
p1 = & a;
p2 = & b;
```
Shameful things to never do

- Not OK:

<table>
<thead>
<tr>
<th>Code Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;anything&gt; = &amp;35;</td>
</tr>
<tr>
<td>&lt;anything&gt; = *35;</td>
</tr>
<tr>
<td>p1 = 35;</td>
</tr>
<tr>
<td>a = &amp;&lt;anything&gt;;</td>
</tr>
<tr>
<td>a = *b;</td>
</tr>
<tr>
<td>*a = &lt;anything&gt;;</td>
</tr>
<tr>
<td>&amp;&lt;anything&gt; = &lt;anything&gt;;</td>
</tr>
<tr>
<td>a = p2;</td>
</tr>
</tbody>
</table>

Initially:

```c
int a, b, *p1, *p2;
a = 30, b = 50;
p1 = &a;
p2 = &b;
```

```c
int a, b, *p1, *p2;
a = **p2;
p1 = b;
p1 = &p2;
p1 = *p2;
<anything> = *b;
*p1 = p2;
*p1 = &<anything>;
```
## Reminder: Precedence of `&` and `*`

<table>
<thead>
<tr>
<th>Tokens</th>
<th>Operator</th>
<th>Class</th>
<th>Prec.</th>
<th>Associates</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>++</code></td>
<td>increment, decrement</td>
<td>prefix</td>
<td></td>
<td>right-to-left</td>
</tr>
<tr>
<td><code>--</code></td>
<td>increment, decrement</td>
<td>prefix</td>
<td></td>
<td>right-to-left</td>
</tr>
<tr>
<td><code>sizeof</code></td>
<td>size</td>
<td>unary</td>
<td></td>
<td>right-to-left</td>
</tr>
<tr>
<td><code>~</code></td>
<td>bit-wise complement</td>
<td>unary</td>
<td></td>
<td>right-to-left</td>
</tr>
<tr>
<td><code>!</code></td>
<td>logical NOT</td>
<td>unary</td>
<td></td>
<td>right-to-left</td>
</tr>
<tr>
<td><code>-</code></td>
<td>negation, plus</td>
<td>unary</td>
<td>15</td>
<td>right-to-left</td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td>address of</td>
<td>unary</td>
<td></td>
<td>right-to-left</td>
</tr>
<tr>
<td><code>*</code></td>
<td>Indirection (dereference)</td>
<td>unary</td>
<td></td>
<td>right-to-left</td>
</tr>
</tbody>
</table>
Pointers as Arguments of Functions

• Pointers can be passed as arguments to functions
• Useful if you want the callee to modify the caller’s variable(s)
  – that is, passing a pointer is the same as passing a reference to (the address of) a variable

• (The pointer itself is passed by value, and the caller’s copy of the pointer cannot be modified by the callee)
As Arguments (cont’d)

```c
void swap ( int * px, int * py ) {
    int temp = *px;
    *px = *py;
    *py = temp;
    px = py = NULL; /* just to show caller’s pointers not changed */
}
```

prints the pointer (not the variable that is pointed to)

```c
int i = 100, j = 500;
int *p1 = &i, *p2 = &j;
printf("%d %d %p %p\n", i, j, p1, p2);
swap(p1, p2);
printf("%d %d %p %p\n", i, j, p1, p2);
```
Exercise 13a

Input and output params

• Write a function that copies the integer src to the memory at pointers dest1 and dest2 unless the pointer in question is NULL. Prototype:
  – void copy2(int src, int* dest1, int* dest2)

• Examples:

```c
int a=0,b=0,c=0;
int* p = &b;

copy2(5,&a,NULL);
printf("%d %d %d\n",a,b,c); // 5 0 0
copy2(a+1,&c,p);
printf("%d %d %d\n",a,b,c); // 5 6 6
copy2(9,NULL,NULL);
printf("%d %d %d\n",a,b,c); // 5 6 6
```
Any Limits on References?

- Like array bounds, in C there are no limitations on what a pointer can address

- Ex:

  ```c
  int *p = (int *) 0x31415926;
  printf("*p = %d\n", *p);
  ```

When I compiled (no errors or warnings) and ran this code, result was:

```
Segmentation fault
```
Pointers as Return Values

• A function can return a pointer as the result

```c
int i, j, *rp;
rp = bigger ( &i, &j );
```

```c
int * bigger ( int *p1, int *p2 )
{
    if (*p1 > *p2)
        return p1;
    else
        return p2;
}
```

Useful? Wouldn't it be easier to return the bigger value (*p1 or *p2)?
...Return Values (cont’d)

• Warning! never return a pointer to an auto variable in the scope of the callee!

• Why not?

```c
int main (void)
{
    printf("%d\n", * sumit ( 3 ));
    printf("%d\n", * sumit ( 4 ));
    printf("%d\n", * sumit ( 5 ));
    return (0);
}

int * sumit ( int i)
{
    int sum = 0;
    sum += i;
    return &sum;
}
```
...Return Values (cont’d)

• But with this change, no problems!

• Why not?

```c
int * sumit ( int i)
{
    static int sum = 0;
    sum += i;
    return &sum;
}
```

Result

3
7
12
```c
int s = 0;
sumit(3, &s); printf("%d\n", s);
sumit(4, &s); printf("%d\n", s);
sumit(5, &s); printf("%d\n", s);
```

```c
void sumit (int i, int *sp)
{
    *sp += i;
    return
}
```
Arrays and Pointers

- An array variable declaration is really two things:
  1. **allocation** (and initialization) of a block of memory large enough to store the array
  2. binding of a **symbolic name** to the address of the start of the array

Ex.: ```
int nums[3] = { 10, 20, 30 };```
Ways to Denote Array Addresses

- Address of first element of the array
  - `nums` (or `nums+0`), or
  - `&nums[0]`
- Address of second element
  - `nums+1`
  - `&nums[1]`
- etc.

Why “+1” and not “+4”? What happened to the “address of” operator?
Arrays as Function Arguments

• Reminder: an array is passed by reference, as an address of (pointer to) the first element

• The following are equivalent

```c
int len, slen ( char s[] );
char str[20] = “a string”;
len = slen(str);
...
int slen(char str[])
{
    int len = 0;
    while (str[len] != ‘\0’)
        len++;
    return len;
}
```

With arrays

```c
int len, slen ( char *s );
char str[20] = “a string”;
len = slen(str);
...
int slen(char *str)
{
    char *strend = str;
    while (*strend != ‘\0’)
        strend++;
    return (strend - str);
}
```

With pointers
Arrays are Pointers

• Ex.: adding together elements of an array

• Version 0, with array indexing:

```c
int i, nums[3] = {10, 20, 30};
int sum = 0;
for (i = 0; i < 3; i++)
    sum += nums[i];
```
...are Pointers (cont’d)

Same example, using pointers (version 1)

```c
int *ap, nums[3] = {10, 20, 30};

int sum = 0;
for (ap = &(nums[0]); ap < &(nums[3]); ap++)
    sum += *ap;
```

- Initialize pointer to starting address of array
- Add next element to sum
- Loop until you exceed the bounds of the array
- Increment pointer to next element in array (pointer arithmetic)
Using pointers in normal way (version 2)

```c
for (ap = nums; ap < (nums+3); ap++)
    sum += *ap;
```

But don’t try to do this

```c
for (ap = (nums+3), nums < ap; nums++)
    sum += *nums;
```
Q: How much is the increment?

A: the size of one element of the array (e.g., 4 bytes for an `int`, 1 byte for a `char`, 8 bytes for a `double`, ...)

```c
int *ap, nums[3] = {10, 20, 30};
int sum = 0;
for (ap = nums; ap <= (nums+2); ap++)
    sum += *ap;
```

```c
char *ap, nums[3] = {10, 20, 30};
char sum = 0;
for (ap = nums; ap <= (nums+2); ap++)
    sum += *ap;
```
...Arithmetic (cont’d)

• Array of **ints**

<table>
<thead>
<tr>
<th>Symbolic Address</th>
<th>Byte Addr</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nums</code></td>
<td>Start of <code>nums</code></td>
<td>10</td>
</tr>
<tr>
<td><code>nums + 1</code></td>
<td>Start of <code>nums + 4</code></td>
<td>20</td>
</tr>
<tr>
<td><code>nums + 2</code></td>
<td>Start of <code>nums + 8</code></td>
<td>30</td>
</tr>
</tbody>
</table>

Array of **chars**

<table>
<thead>
<tr>
<th>Symbolic Address</th>
<th>Byte Addr</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nums</code></td>
<td>Start of <code>nums</code></td>
<td>10</td>
</tr>
<tr>
<td><code>nums + 1</code></td>
<td>Start of <code>nums + 1</code></td>
<td>20</td>
</tr>
<tr>
<td><code>nums + 2</code></td>
<td>Start of <code>nums + 2</code></td>
<td>30</td>
</tr>
</tbody>
</table>
...Arithmetic (cont’d)

• Referencing the ith element of an array

\[
\text{int } \text{nums}[10] = \{\ldots\}; \\
\ldots \\
\text{nums}[i-1] = 50; \\
\text{int } \text{nums}[10] = \{\ldots\}; \\
\ldots \\
*(\text{nums} + i - 1) = 50;
\]

Equivalent

Referencing the end of an array

\[
\text{int } *\text{np}, \text{nums}[10] = \{\ldots\}; \\
\ldots \\
\text{for } (\text{np} = \text{nums}; \text{np} < (\text{nums}+10); \text{np}++) \\
\ldots
\]
A Special Case of Array Declaration

• Declaring a pointer to a string literal also allocates the memory containing that string

• Example:

```c
char *str = "This is a string";
```

is equivalent to...

```c
char str[] = "This is a string";
```

Except! first version is read only (cannot modify string contents in your program)!

 Doesn’t work with other types or arrays, ex.:

```c
int *nums = {0, 1, 2, 3, 4}; // won’t work!
char *str = {‘T’,‘h’,‘i’,‘s’}; // no NULL char
```
Input Arguments to `scanf()` again

• Must be passed using “by reference”, so that `scanf()` can overwrite their value
  – arrays, strings: just specify array name
  – anything else: pass a pointer to the argument

• Ex.:

```c
char c, str[10];
int j;
double num;
int result;

result =
    scanf("%c %9s %d %lf", &c, str, &j, &num);
```
Multidimensional Arrays and Pointers

- 2-D array ≡
  
  1-D array of 1-D arrays

```c
double rain[years][months] = {
  {3.1, 2.6, 4.3, ...},
  {2.7, 2.8, 4.1, ...},
  ...
};
```

```c
test = 3, month = 5;
rain[test][month] = 2.4;
```

```c
double *yp, *mp;
yp = rain[3];
mp = yp + 5;
*mp = 2.4;
```

Remember:
- `rain` is the address of the entire array
- `rain[3]` is the address of the 4th row of the array
- `rain[3][5]` is the value of the 6th element in the 4th row
- `&rain[3][5]` is the address of the 6th element in the 4th row
- `yp` = address of 4th row
- `mp` = address of 6th element in 4th row
...Multidimensional (cont’d)

- Equivalent:

```c
double *yp, *mp;
yp = rain[3];
mp = yp + 5;
*mp = 2.4;
```

Remember:
- `rain` is the address of the entire array
- `rain[3]` is the address of the 4th row of the array
- `rain[3][5]` is the value of the 6th element in the 4th row
- `&(rain[3][5])` is the address of the 6th element in the 4th row

```c
double *mp;
mp = &(rain[3][5]);
*mp = 2.4;
```
2-D Array of Equal Length Strings

- Ex. using indexing

```c
char strings[4][7] = {
    "Blue", "Green", "Orange", "Red"
};
...
printf ("%s\n", strings[3]);
int i = 0;
strings[2][i++] = 'W', strings[2][i++] = 'h',
strings[2][i++] = 'i', strings[2][i++] = 't',
strings[2][i++] = 'e', strings[2][i++] = '\0';

printf ("%c\n", strings[2][3]);
```
...Equal Length Strings (cont’d)

• With pointers

```c
char strings[4][7] = {
    "Blue", "Green", "Orange", "Red"
};
...
printf ("%s\n", *(strings+3));
char *cp = strings[2];
*cp++ = 'W', *cp++ = 'h', *cp++ = 'i',
*cp++ = 't', *cp++ = 'e', *cp++ = '\0';

cp = strings[2];
printf ("%c\n", *(cp+3));
```
Equal Length Strings In Memory

<table>
<thead>
<tr>
<th>Blue\0\0\0</th>
<th>Green\0\0</th>
<th>Orange\0</th>
<th>Red\0\0\0\0</th>
</tr>
</thead>
</table>

8 bytes  8 bytes  8 bytes  8 bytes
2-D Array of Unequal Length Strings

- Example, using array indexing

```c
char *strings[4] =
{ "Blue", "Green", "Orange", "Red" };

printf("%s\n", strings[3]);
for (i = 0; i < 4; i++) {
    int len = 0;
    for (j = 0; strings[i][j] != '\0'; j++)
        len += 1;
    printf("length %d = %d\n", i, len);
}
printf("%c\n", *(strings[2]+3));
```

`strings[]` is both a 1-D array of pointers to strings and a 2-D array of characters!
Unequal Length Strings In Memory

Less storage?

- 5 bytes
- 6 bytes
- 7 bytes
- 4 bytes

Blue\0 Green\0 Orange\0 Red\0


8 bytes 8 bytes 8 bytes 8 bytes

• (don’t forget there is storage for the pointers)
...Unequal (cont’d)

- Ex., using pointers

    char *strings[4] = 
    { "Blue", "Green", "Orange", "Red" }; 
    char *cp = strings[3]; 
    printf ("%s\n", cp); 
    for (int i = 0; i < 4; i++) {
        int len = 0;
        cp = strings[i]; 
        while (*cp++ != '\0') 
            len += 1;
        printf("length %d = %d\n", i, len);
    }
    cp = strings[2] + 3;
    printf("%c\n", *cp);
structs Containing Pointers

- structs are groups of fields into a single, named record (similar to an object)
- Lots of uses, e.g., linked lists

More about this when we discuss structs
Pointers to Functions

- Another level of indirection: which function you want to execute
- Example: giving raises to employees
  - Type A employee gets $5000 raise, type B get $8000
- Two ways to do it
  1. caller tells callee *how much raise* to give
  2. caller tells callee *what function to call* to get the amount of the raise
float sals[NUMOFEMPLOYEES];
void raise (int empnum, int incr);
...
int emp1 = ...;
raise ( emp1, 5000 );
...
void raise (int empid, int incr)
{
    sals[empid] += incr; /* give the employee * a raise */
}
Approach #2

float sals[NUMOFEMPLOYEES];
void raise (int, int () );
int raiseTypeA ( int );
int raiseTypeB ( int );

int emp1 = ...;
raise ( emp1, raiseTypeA );
...

void raise ( int empid, int raiseType () )
{
    sals[empid] += raiseType (empid);
}
...

int raiseTypeA (int eid) { ... };
int raiseTypeB (int eid) { ... };

Points to Functions (cont’d)

- Another type of input parameter

```c
void raise (int, int () );
```

or...

```c
void raise (int empid, int (*rt) () );
```

A function name used as an argument is a pointer to that function
- & and * are not needed!

You cannot modify a function during execution; you can only modify the pointer to a function

Advantages to approach #1? approach #2?
A Better Example

- Standard library function for **sorting**:

```c
void qsort( void *base, size_t n, size_t size, int (*cmp)( const void *, const void * ) ) ;
```

- **starting address of array to be sorted** (in place)
- **number of objects to sort**
- **size of each object**
- Function that compares two objects and returns < 0, 0, or > 0 if object 1 is < object 2, == object 2, or > object 2, resp.

Why is it **necessary** to pass a pointer to a function in this case?
Any Questions?