

Pointers in C

C Programming and Software Tools

N.C. State Department of Computer Science

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"
- "w = the value of the int whose address is contained in pv"

Same as this!

```
int v, w;  
int *pv;  
  
pv = &v;  
w = *pv;
```

CSC230: C and Software Tools © NC State Computer Science Faculty

C Pointer Operators

<code>px = &x;</code>	"px is assigned the address of x"
<code>y = *px;</code>	"y is assigned the value at the address indicated (pointed to) by px"

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

CSC230: C and Software Tools © NC State Computer Science Faculty

...Operators (cont'd)

- **&** = "the address of..."

```
int a;  
int *ap;
```

"ap is a pointer
to an int"

```
ap = &a;
```

"ap gets the address
of variable a"

```
char c;  
char *cp;
```

"cp is a pointer
to a char"

```
cp = &c;
```

"cp gets the address
of variable c"

```
float f;  
float *fp;
```

"fp is a pointer
to a float"

```
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)"

Variable Names Refer to Memory

- A C expression, **without** pointers

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

Symbol Table

Memory Address	Variable
0	b
4	c
8	a

"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
```

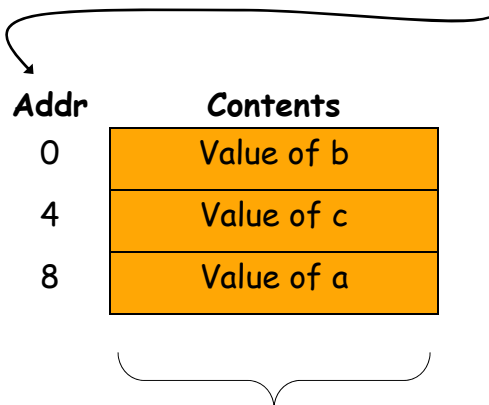
CSC230: C and Software Tools © NC State

Computer Science
NC STATE UNIVERSITY

9

Variables Stored in Memory

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



32 bits (4 bytes) wide

Computer Science
NC STATE UNIVERSITY

CSC230: C and Software Tools © NC State

10

Pointers Refer to Memory Also

- A C expression, **with** pointers

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

Memory Address	Variable
0	b
4	c
8	a
12	ap

"Pseudo-assembler" code

```
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
```

CSC230: C and Software Tools © NC State

Computer Science
NC STATE UNIVERSITY

11

Pointers Refer... (cont'd)

Address	Contents	Variable Name
0	Value of b	b
4	Value of c	c
8	Value of a	a
12	8 (address of a)	ap

32 bits (4 bytes) wide

CSC230: C and Software Tools © NC State

Computer Science
NC STATE UNIVERSITY

12

Addresses vs. Values

```
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
```

???

Pointers to Pointers to ...

- A C expression

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

Var	Address
a	8
ap	12
app	20
appp	16
b	0
c	4

Addr	Contents	Var
0	Value of b	b
4	Value of c	c
8	Value of a	a
12	8 (addr of a)	ap
16	20 (addr of app)	appp
20	12 (addr of ap)	app

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

```
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

Computer Science
NC STATE UNIVERSITY

CSC230: C and Software Tools © NC State Computer Science Faculty

17

Pointer Type Conversions

- **Pointer casts** are possible, but **rarely (never?) useful**

```
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!

Computer Science
NC STATE UNIVERSITY

CSC230: C and Software Tools © NC State Computer Science Faculty

18

...Conversions (cont'd)

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

```
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?

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

1. `ap = &c;`

2. `*ap = 3333;`

3. `c = ap;`

4. `c = *ap;`

⚠ common source of bugs ⚠
**pretty much
everything
to do with pointers**

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

Bloopers (cont'd)

```
5. dp = ap;
```

```
6. dp = 'Q';
```

```
7. fp = 3.14159;
```

```
8. gp = &fp;
```

```
9. *gp = 3.14159;
```

CSC230: C and Software Tools © NC State Computer Science Faculty

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

... Bloopers (cont'd)

```
10. *fp = &gp;
```

```
11. &gp = &fp;
```

```
12. b = *a;
```

```
13. b = &a;
```

CSC230: C and Software Tools © NC State Computer Science Faculty

Sense...

Initially:

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

• OK:


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

...and Nonsensibility


Initially:

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

• Not OK:



<code><anything> = &35;</code>
<code><anything> = *35;</code>
<code>p1 = 35;</code>
<code>a = &<anything>;</code>
<code>a = *b;</code>
<code>*a = <anything>;</code>
<code>&<anything> = <anything>;</code>
<code>a = p2;</code>



<code>a = **p2;</code>
<code>p1 = b;</code>
<code>p1 = &p2;</code>
<code>p1 = *p2;</code>
<code><anything> = *b;</code>
<code>*p1 = p2;</code>
<code>*p1 = &<anything>;</code>

Reminder: Precedence of & and *

Tokens	Operator	Class	Prec.	Associates
++ --	increment, decrement	prefix	15	right-to-left
sizeof	size	unary		right-to-left
~	bit-wise complement	unary		right-to-left
!	logical NOT	unary		right-to-left
- +	negation, plus	unary		right-to-left
&	address of	unary		right-to-left
*	Indirection (dereference)	unary		right-to-left

Pointers as Arguments of Functions

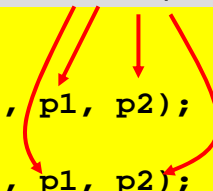
- Pointers can be passed as arguments to functions
- Useful if you want the callee to modify the caller's variable(s)
 - that is, passing a pointer is the same as passing a reference to (the address of) a variable
- (The pointer itself is passed by value, and the caller's copy of the pointer cannot be modified by the callee)

...as Arguments (cont'd)

```
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)

```
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);
```



CSC230: C and Software Tools © NC State Computer Science Faculty

30 NC STATE UNIVERSITY

...as Arguments (cont'd)

- Results of execution:

CSC230: C and Software Tools © NC State Computer Science Faculty

Computer Science
31 NC STATE UNIVERSITY

Any Limits on References?

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

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

who knows what is
stored at this location?!

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

Segmentation fault

Computer Science
32 NC STATE UNIVERSITY

CSC230: C and Software Tools © NC State Computer Science Faculty

Pointers as Return Values

- A function can **return** a pointer as the result

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

```
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) ?

Computer Science
33 NC STATE UNIVERSITY

CSC230: C and Software Tools © NC State Computer Science Faculty

...Return Values (cont'd)

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

- Why not?

```
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;
}
```

CSC230: C and Software Tools © NC State Computer Science Faculty

34 NC STATE UNIVERSITY

...Return Values (cont'd)

But with
this
change, no
problems!

Why not?

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

Result

```
3
7
12
```

CSC230: C and Software Tools © NC State Computer Science Faculty

Computer Science
35 NC STATE UNIVERSITY

Alternative...

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

```
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 };`

Byte Address	Contents	Block of Memory
nums	10	
nums + 4	20	
nums + 8	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**

```
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**

```
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 R' Pointers

- Ex.: adding together elements of an array
- Version 0, with array **indexing**:

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

...R' Pointers (cont'd)

Same example, using **pointers** (version 1)

pointer to int

increment pointer to
next element in array
(pointer arithmetic)

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

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

add next element to sum

initialize pointer to
starting address of array

loop until you exceed the
bounds of the array

...R' Pointers (cont'd)

Using **pointers** in normal way (version 2)

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

initialize pointer to
starting address of array

loop until you exceed the
bounds of the array -
more pointer arithmetic

But **don't** try to do this

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

CSC230: C and Software Tools © NC State Computer Science Faculty

42

NC STATE UNIVERSITY

Pointer Arithmetic

- Q: How **much** is the increment?

Add 4 to the address

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

Add 1 to the address

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

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, ...)

CSC230: C and Software Tools © NC State Computer Science Faculty

43

NC STATE UNIVERSITY

...Arithmetic (cont'd)

- Array of **ints**

Symbolic Address	Byte Addr	Contents
<code>nums</code>	Start of <code>nums</code>	10
<code>nums+1</code>	Start of <code>nums</code> + 4	20
<code>nums+2</code>	Start of <code>nums</code> + 8	30

- Array of **chars**

Symbolic Address	Byte Addr	Contents
<code>nums</code>	Start of <code>nums</code>	10
<code>nums+1</code>	Start of <code>nums</code> + 1	20
<code>nums+2</code>	Start of <code>nums</code> + 2	30

CSC230: C and Software Tools © NC State Computer Science Faculty

44

Computer Science
NC STATE UNIVERSITY

...Arithmetic (cont'd)

- Referencing the *i*th element of an array

```
int nums[10] = {...};  
...  
nums[i-1] = 50;
```

```
int nums[10] = {...};  
...  
*(nums + i - 1) = 50;
```

Equivalent

Referencing the end of an array

```
int *np, nums[10] = {...};  
...  
for (np = nums; np < (nums+10); np++)  
...  
}
```

CSC230: C and Software Tools © NC State Computer Science Faculty

45

Computer Science
NC STATE UNIVERSITY

A Special Case of Array Declaration

- Declaring a pointer to a **string literal** also allocates the memory containing that string
- Example:

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

is equivalent to...

```
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.:

```
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 “call by reference”, so that **scanf()** can overwrite their value
 - arrays, strings: just specify array name
 - anything else: pass a **pointer** to the argument
- Ex.:

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

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

⚠ common source of bugs ⚠
failure to use &
before arguments
to scanf

Multidimensional Arrays and Pointers

- 2-D array \equiv
1-D array of 1-D arrays

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

```
year = 3, month = 5;  
rain[year][month] = 2.4;
```

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

CSC230: C and Software Tools ©

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:

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

```
double *mp;  
mp = &(rain[3][5]);  
*mp = 2.4;
```

inconsistent?

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

2-D Array of Equal Length Strings

- Ex. using **indexing**

4 rows, each with 7 characters
(i.e., each row is a string)

```
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]);
```

CSC230: C and Software Tools © NC State Computer Science Faculty

50

NC STATE UNIVERSITY

Equal Length Strings... (cont'd)

- Result

```
Red  
t
```

CSC230: C and Software Tools © NC State Computer Science Faculty

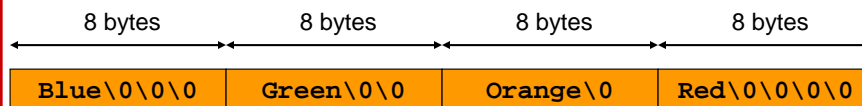
Computer Science
51 NC STATE UNIVERSITY

...Equal Length Strings (cont'd)

- With **pointers**

```
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



2-D Array of Unequal Length Strings)

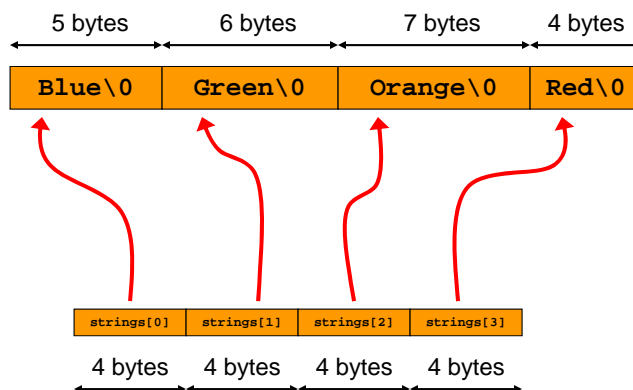
- Ex., using **array indexing**

```
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?



- (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);
```

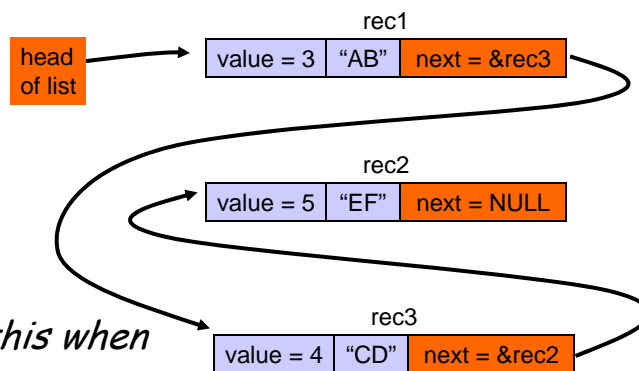
CSC230: C and Software Tools © NC State Computer Science Faculty

56

Computer Science
NC STATE UNIVERSITY

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*

CSC230: C and Software Tools © NC State Computer Science Faculty

57

Computer Science
NC STATE UNIVERSITY

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

Approach #1

```
float sals[NUMOFEMPLOYEES];
void raise (int empnum, int incr );
...
int empl = ...;
(void) raise ( empl, 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 empl = ...;
(void) raise ( empl, raiseTypeA );
...
void raise ( int empid, int raiseType () )
{
    sals[empid] += raiseType (empid);
}
...
int raiseTypeA (int eid) { ... };
int raiseTypeB (int eid) { ... };
```

CSC230: C and Software Tools © NC State Computer Science Faculty

60

Pointers to Functions (cont'd)

- Another type of input parameter

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

or...

```
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?

Computer Science
61 NC STATE UNIVERSITY

CSC230: C and Software Tools © NC State Computer Science Faculty

A Better Example

- Standard library function for **sorting**:

```
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?