Pointers in C

ITCS 2116: C Programming

College of Computing and Informatics

Department of Computer Science

UNIVERSITY OF NORTH CAROLINA
CHARLOTTE

Part 1

Pointer Variables

Pointers in Every Day Life

- Examples
  - telephone numbers
  - web pages
  - Twitter ID
- Principle: **indirection**

> You can call someone or send them a text message using their phone number.
>
> The phone number may change over time, but the message or call can still reach the same person.
>
> One phone number may be forwarded to another phone number, which may be forwarded to another, and so on.

- This approach has many benefits, such as:
  - Enables dynamic memory allocation
  - Makes it possible to implement data structures (e.g., linked lists)

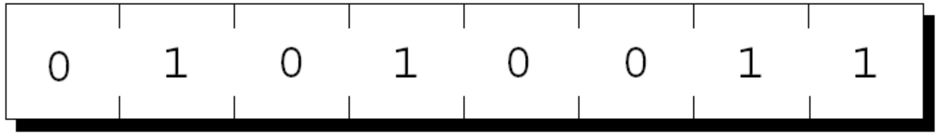
UNIVERSITY OF NORTH CAROLINA CHARLOTTE

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 $\rightarrow$ address translation)
- You **could** just write code that uses addresses (no symbolic names)
  – advantages? disadvantages?

Pointer Variables

- The first step in understanding pointers is visualizing what they represent at the machine level.
- In most modern computers, main memory is divided into ***bytes,*** with each byte capable of storing eight bits of information:



- Each byte has a unique ***address.***

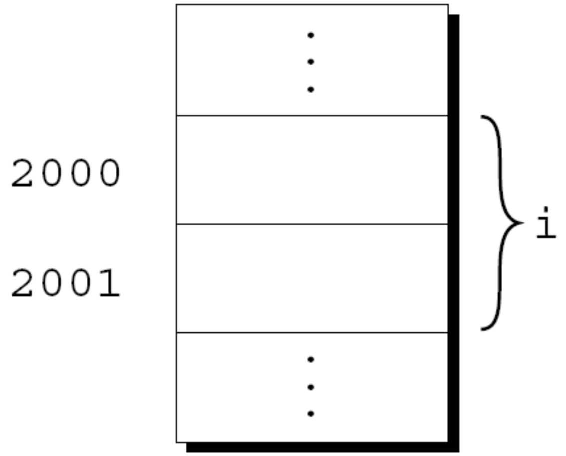
Pointer Variables

If there are *n* bytes in memory, we can think of addresses as numbers that range from 0 to *n* – 1:

Address	Contents
0	01010011
1	01110101
2	01110011
3	01100001
4	01101110
	⋮
n-1	01000011

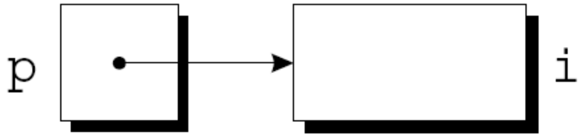
Pointer Variables

- Each variable in a program occupies one or more bytes of memory.
- The address of the first byte is said to be the address of the variable.
- In the following figure, the address of the variable **i** is 2000:



Pointer Variables

- Addresses can be stored in special ***pointer variables.***
- When we store the address of a variable **`i`** in the pointer variable **`p`**, we say that **`p`** "points to" **`i`**.
- A graphical representation:



Declaring Pointer Variables

- When a pointer variable is declared, its name must be preceded by an asterisk:

```
int *p;
```

- **`p`** is a pointer variable capable of pointing to ***objects*** of type **`int`**.
- We use the term *object* instead of *variable* since **`p`** might point to an area of memory that doesn't belong to a variable.

Declaring Pointer Variables

- Pointer variables can appear in declarations along with other variables:

```
int i, j, a[10], b[20], *p, *q;
```

- C requires that every pointer variable point only to objects of a particular type (the ***referenced type***):

```
int *p;      /* points only to integers   */
double *q;   /* points only to doubles    */
char *r;     /* points only to characters */
```

- There are no restrictions on what the referenced type may be.

The Address and Indirection Operators

- C provides a pair of operators designed specifically for use with pointers.
  - To find the address of a variable, we use the **&** (**address of**) operator.
  - To gain access to the object that a pointer points to, we use the ***** (***indirection***) operator.

The Address Operator

- Declaring a pointer variable sets aside space for a pointer but doesn't make it point to an object:

```
int *p;  /* points nowhere in particular */
```

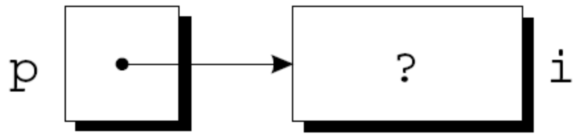
- It's crucial to **initialize** `p` before we use it.
- Trying to use a pointer that has not been initialized will usually result in program failure.

The Address Operator

- One way to initialize a pointer variable is to assign it the address of a variable:

```
int i, *p;
…
p = &i;
```

- Assigning the address of **`i`** to the pointer variable **`p`** makes **`p`** point to **`i`**:



The Address Operator

- It’s also possible to initialize a pointer variable at the time it’s declared:

```
int i;
int *p = &i;
```

- The declaration of **i** can even be combined with the declaration of **p**:

```
int i, *p = &i;
```

The Indirection Operator

- Once a pointer variable points to an object, we can use the * (**indirection**) operator to access what is stored in the object.
- If p points to i, we can print the value of i as follows:

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

- Applying & to a variable produces a pointer to the variable.
- Applying * to the pointer takes us back to the original variable:

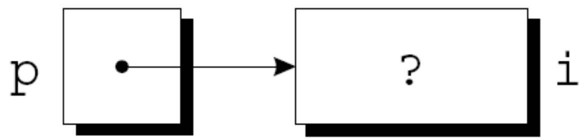
```
j = *&i;    /* same as j = i; */
```

The Indirection Operator

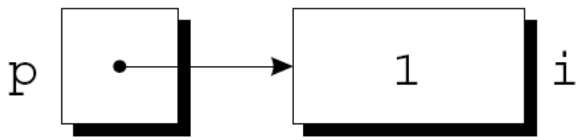
- As long as `p` points to `i`, `*p` is an ***alias*** for `i`.
  - The expression `*p` has the same value as `i`.
  - Changing the value of `*p` changes the value of `i`.
- The example on the next slide illustrates the equivalence of `*p` and `i`.

The Indirection Operator

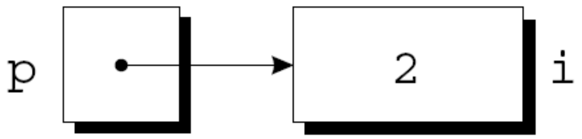
```
p = &i;
```




```
i = 1;
```




```
printf("%d\n", i);     /* prints 1 */
printf("%d\n", *p);    /* prints 1 */
*p = 2;
```




```
printf("%d\n", i);     /* prints 2 */
printf("%d\n", *p);    /* prints 2 */
```

The Indirection Operator

- Applying the indirection operator to an uninitialized pointer variable causes undefined behavior:

```
int *p;
printf("%d", *p);    /*** WRONG ***/
```

- Assigning a value to **`*p`** is particularly dangerous:

```
int *p;
*p = 1;    /*** WRONG ***/
```

Pointer Assignment

- C allows the use of the assignment operator to copy pointers of the same type.
- Assume that the following declaration is in effect:

```
int i, j, *p, *q;
```

- Example of pointer assignment:

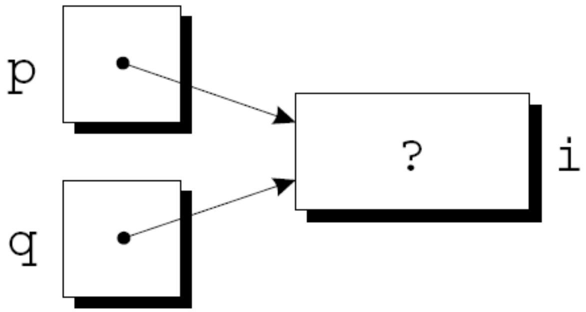
```
p = &i;
```

Pointer Assignment

- Another example of pointer assignment:

  ```
  q = p;
  ```

  `q` now points to the same place as **`p`**:

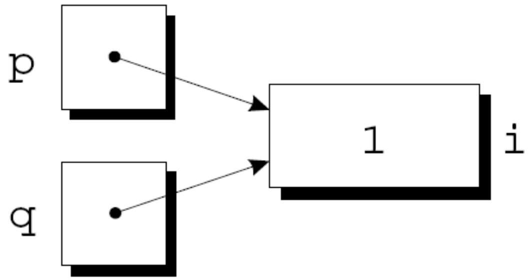


UNIVERSITY OF NORTH CAROLINA CHARLOTTE

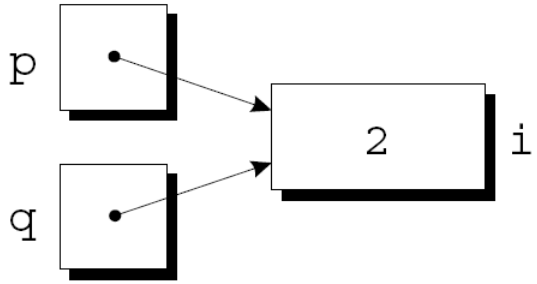
Pointer Assignment

- If **p** and **q** both point to **i**, we can change **i** by assigning a new value to either ***p** or ***q**:

```
*p = 1;
```




```
*q = 2;
```



- Any number of pointer variables may point to the same object.

Pointer Assignment

- Be careful not to confuse

  ```
  q = p;
  ```

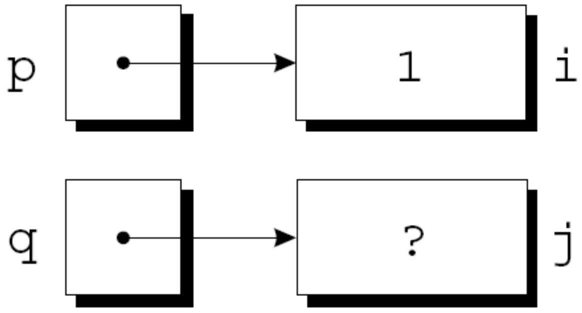
  with

  ```
  *q = *p;
  ```

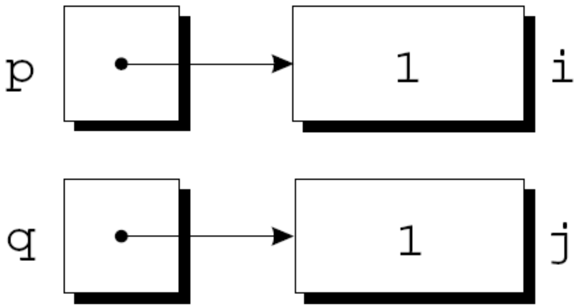
- The first statement is a pointer assignment, but the second is not.
- The example on the next slide shows the effect of the second statement.

Pointer Assignment

```
p = &i;
q = &j;
i = 1;
```




```
*q = *p;
```



Pointer Operations in C

Consider the code below

- "**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**"

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

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


UNIVERSITY OF NORTH CAROLINA CHARLOTTE

C Pointer Operators

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

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

…Operators (cont'd)

& = "the address of…"

```
int a;
int *ap;

ap = &a;
```

"**ap** is a pointer to an **int**"

"**ap** gets the address of variable **a**"

```
char c;
char *cp;

cp = &c;
```

"**cp** is a pointer to a **char**"

"**cp** gets the address of variable c"

```
float f;
float *fp;

fp = &f;
```

"**fp** is a pointer to a **float**"

"**fp** gets the address of variable **f**"

UNIVERSITY OF NORTH CAROLINA CHARLOTTE

…Operators (cont'd)

***** = "pointer to…" or *indirection* operator

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

"the variable ap points to (i.e., a), is assigned the value 33"

"b is assigned the value of the variable pointed to by ap"(i.e., a)

the value of b is 33

```
*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 the 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
```

Variables Stored in Memory

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

Addr	Contents
0	Value of b
4	Value of c
8	Value of a

32 bits (**4 bytes**) wide

Pointers Refer to Memory Also

A C expression, **with** pointers

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

Symbol Table

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
```

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

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
```

???

This code produces compiler warnings – to avoid compiler warnings use the **%p** format specifier.

%p allows us to print the value of a pointer, i.e., the memory address it contains.

(see **addresses_values.c** in *Code samples and Demonstrations* in *Canvas*).

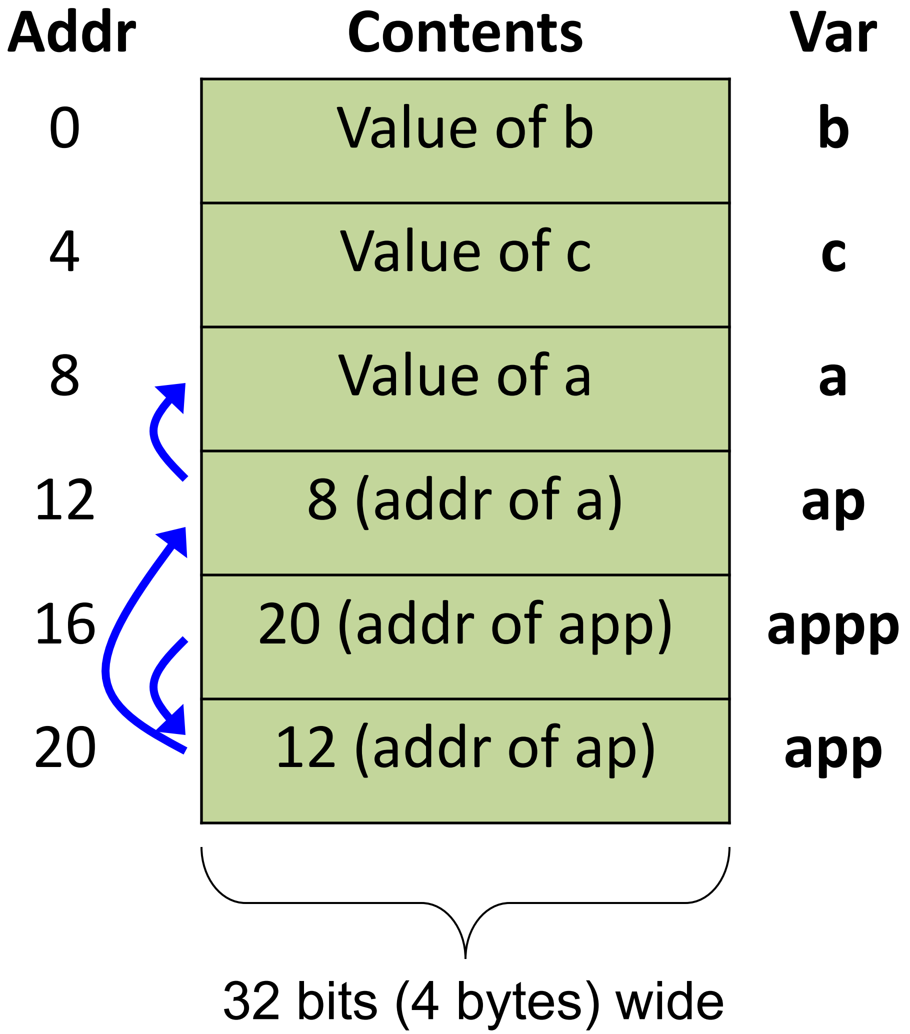
Pointers to Pointers to …

A C expression

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

(see `ptrs_to.c` in *Code samples and Demonstrations* in *Canvas*).

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



…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 */
```

Example: 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 (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!

…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! */
```

Pointer Mistakes

The following slides show examples of **common mistakes** programmers make when using pointers in C.

Find the Pointer Mistakes

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

Find the Pointer Mistakes

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;
```

incompatible types

```
2.   *ap = 3333;
```

```
3.   c = ap;
```

incompatible types

```
4.   c = *ap;
```

… Mistakes (cont'd)

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

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

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

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

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

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


UNIVERSITY OF NORTH CAROLINA CHARLOTTE

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

… Mistakes (cont'd)

```
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;
```

… Mistakes (cont'd)

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

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

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

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

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

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

… Mistakes (cont’d)

```
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

Sense…

Initially:

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

All of these are OK

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

(see `sense.c` in Code samples and Demonstrations in Canvas)

…and Nonsensibility

Initially:

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

None of these are OK

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

(see **nonsensibility.c** in Code samples and Demonstrations in Canvas)

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);
```


UNIVERSITY OF NORTH CAROLINA CHARLOTTE

…as Arguments (cont'd)

- Results of execution: ???
- Download `arguments.c` from *Canvas (Code samples and Demonstrations)*, execute it and examine the output.

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`) ?

…Return Values (cont'd)

- Warning! **never return a pointer to an `auto` variable** in the scope of the callee!
- Why not?
  - Because an **`auto`** variable has no scope outside of the function.

```
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;
}
```

(see **`sumit.c`** in *Code samples and Demonstrations* in *Canvas*).

UNIVERSITY OF NORTH CAROLINA CHARLOTTE

…Return Values (cont'd)

But with this change, no problems!

Why not?

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

Output:

```
3
7
12
```

Download **sumit.c** from *Canvas*, execute it and examine the output.

UNIVERSITY OF NORTH CAROLINA CHARLOTTE

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
}
```

Use a pointer to the variable that you want to contain the sum instead. That variable can remain local to the caller without running into scope issues.

References

- K. N. King, *C Programming: A Modern Approach*, 2nd Edition. W. W. Norton & Company. 2008.
- D.S. Malik, *C++ Programming: From Problem Analysis to Program Design*, Seventh Edition. Cengage Learning. 2014.

UNIVERSITY OF NORTH CAROLINA CHARLOTTE