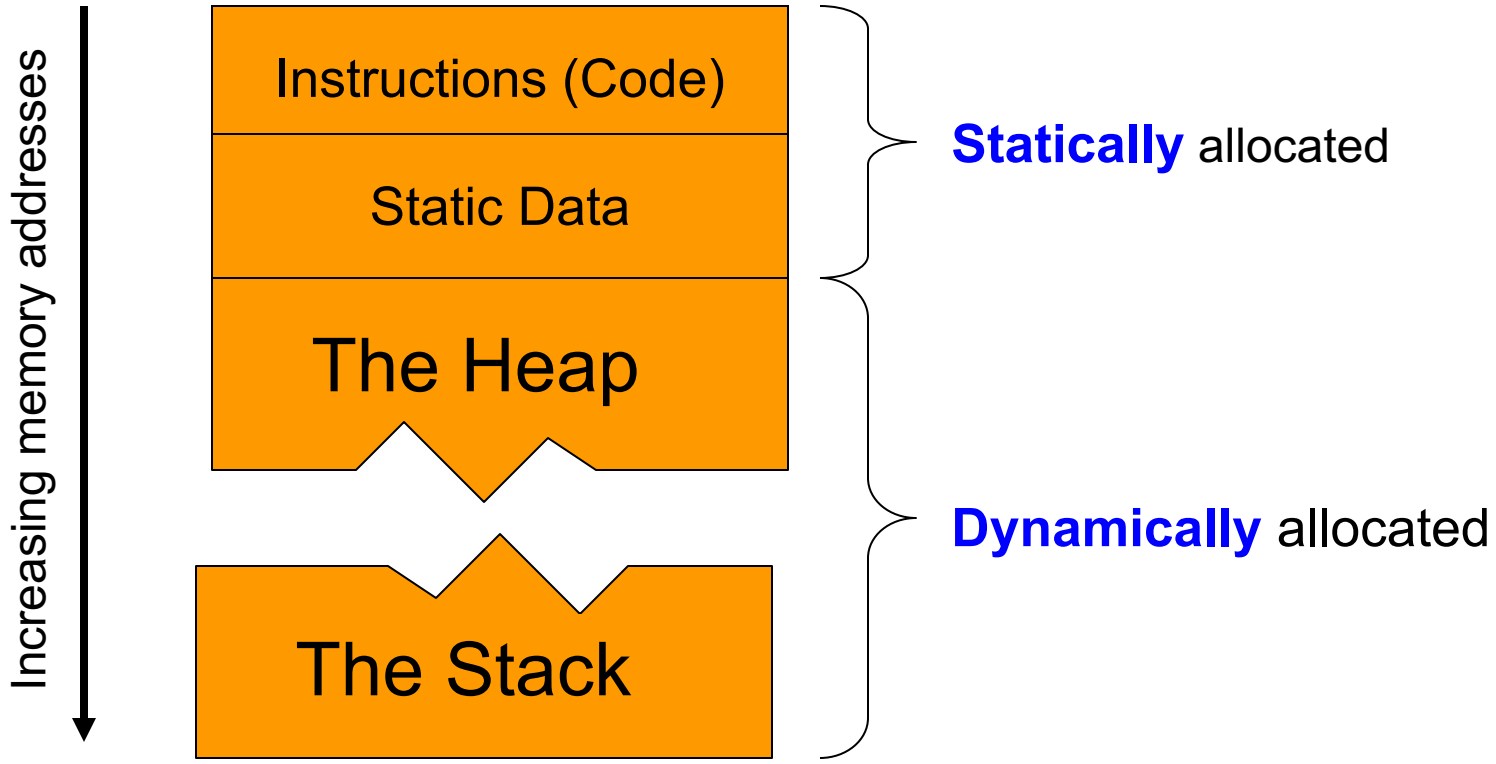
Pointers Variables in C

ITSC 2181: Introduction to Computer Systems
UNC Charlotte
College of Computing and Informatics

COLLEGE OF COMPUTING AND INFORMATICS

Memory Layout of a Program



COLLEGE OF COMPUTING AND INFORMATICS

Pointers in Every Day Life

- Examples
  - telephone numbers
  - web pages
  - Twitter/X 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)

COLLEGE OF COMPUTING AND INFORMATICS

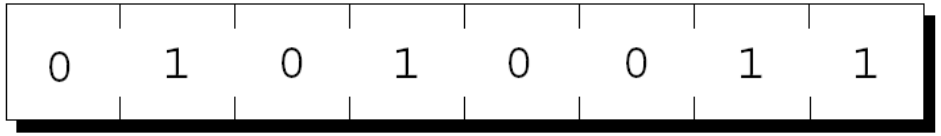
All References are Addresses?

- 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?

COLLEGE OF COMPUTING AND INFORMATICS

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:

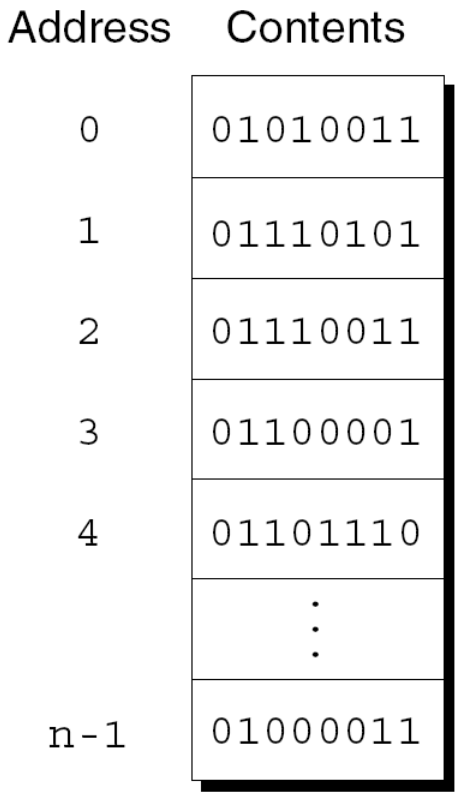


- Each byte has a unique ***address.***

COLLEGE OF COMPUTING AND INFORMATICS
Copyright © 2008 W. W. Norton & Company. All rights reserved.

Pointer Variables

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

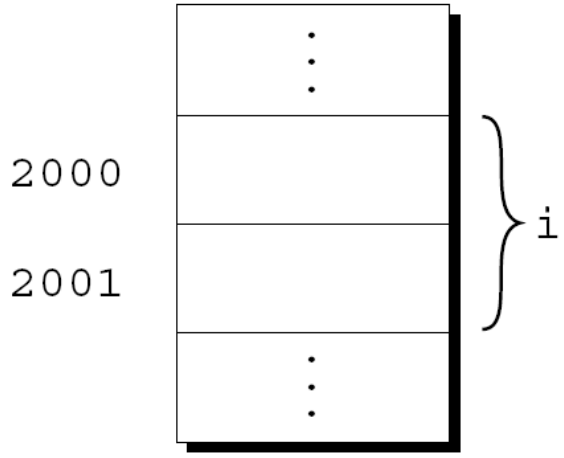


Copyright © 2008 W. W. Norton & Company. All rights reserved.

COLLEGE OF COMPUTING AND INFORMATICS

Pointer Variables

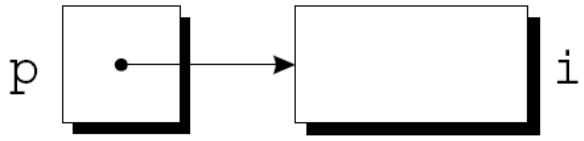
- Each variable in a program occupies one or more bytes of memory.
- The address of the first byte is said to be the address of the variable.
- In the following figure, the address of the variable `i` is 2000:



Copyright © 2008 W. W. Norton & Company. All rights reserved.
COLLEGE OF COMPUTING AND INFORMATICS

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:



Copyright © 2008 W. W. Norton & Company. All rights reserved.
COLLEGE OF COMPUTING AND INFORMATICS

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.

Copyright © 2008 W. W. Norton & Company. All rights reserved.
COLLEGE OF COMPUTING AND INFORMATICS

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.

Copyright © 2008 W. W. Norton & Company. All rights reserved.
COLLEGE OF COMPUTING AND INFORMATICS

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.

Copyright © 2008 W. W. Norton & Company. All rights reserved.

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.

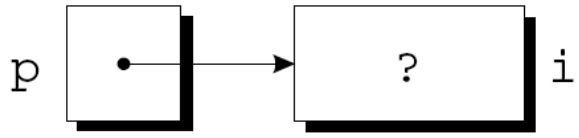
Copyright © 2008 W. W. Norton & Company. All rights reserved.
COLLEGE OF COMPUTING AND INFORMATICS

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`**:



Copyright © 2008 W. W. Norton & Company. All rights reserved.
COLLEGE OF COMPUTING AND INFORMATICS

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;
```


Copyright © 2008 W. W. Norton & Company. All rights reserved.
COLLEGE OF COMPUTING AND INFORMATICS

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; */
```


Copyright © 2008 W. W. Norton & Company. All rights reserved.
COLLEGE OF COMPUTING AND INFORMATICS

The Indirection Operator

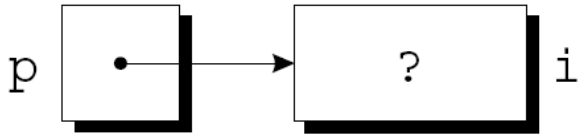
- As long as `p` points to `i`, `*p` is an ***alias*** for `i`.
  - The expression `*p` has the same value as `i`.
  - Changing the value of `*p` changes the value of `i`.
- The example on the next slide illustrates the equivalence of `*p` and `i`.

We use of the indirection operator (`*`) to access the data at the address stored in the pointer.

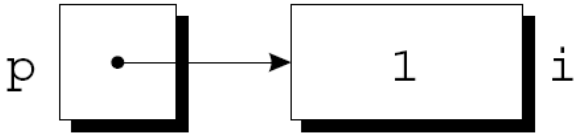
Copyright © 2008 W. W. Norton & Company. All rights reserved.

The Indirection Operator

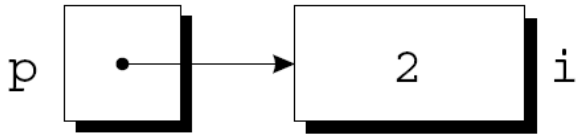
```
p = &i;
```




```
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 */
```


Copyright © 2008 W. W. Norton & Company. All rights reserved.

COLLEGE OF COMPUTING AND INFORMATICS

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 ***/
```


Copyright © 2008 W. W. Norton & Company. All rights reserved.
COLLEGE OF COMPUTING AND INFORMATICS

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;
```

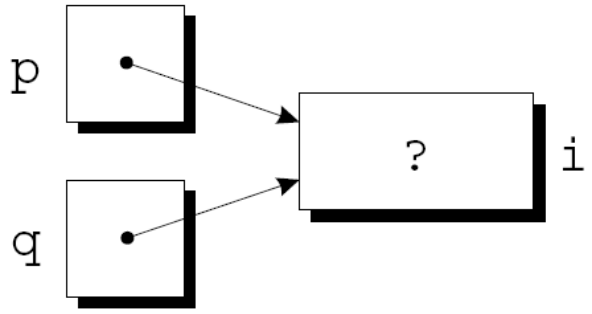

Copyright © 2008 W. W. Norton & Company. All rights reserved.
COLLEGE OF COMPUTING AND INFORMATICS

Pointer Assignment

- Another example of pointer assignment:

  ```
  q = p;
  ```

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

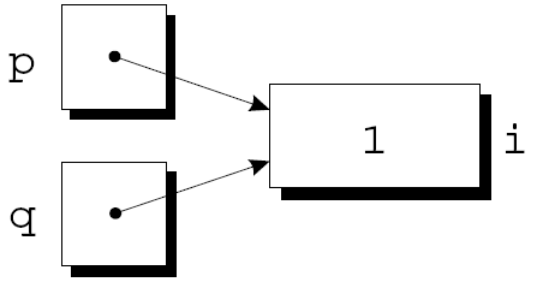


Copyright © 2008 W. W. Norton & Company. All rights reserved.
COLLEGE OF COMPUTING AND INFORMATICS

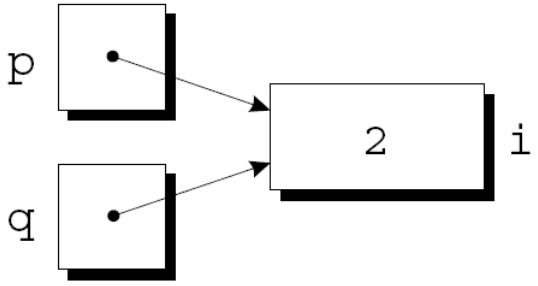
Pointer Assignment

- 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.

Copyright © 2008 W. W. Norton & Company. All rights reserved.
COLLEGE OF COMPUTING AND INFORMATICS

Pointer Assignment

- Be careful not to confuse

  ```
  q = p;
  ```

  with

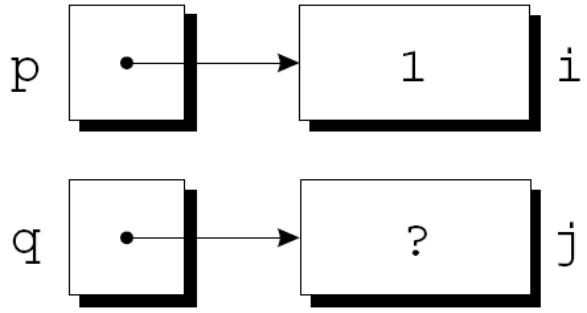
  ```
  *q = *p;
  ```

- 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.

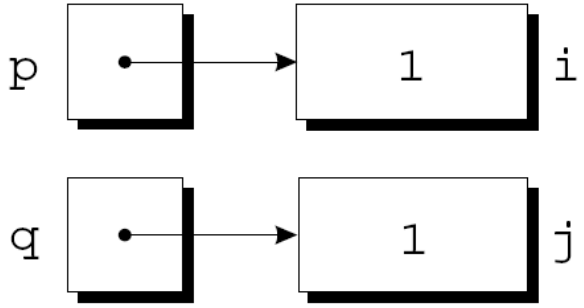
Copyright © 2008 W. W. Norton & Company. All rights reserved.
COLLEGE OF COMPUTING AND INFORMATICS

Pointer Assignment

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




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



Copyright © 2008 W. W. Norton & Company. All rights reserved.

COLLEGE OF COMPUTING AND INFORMATICS

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;
```

(see **test.c** in *Code Samples and Demonstrations* in *Canvas*)

COLLEGE OF COMPUTING AND INFORMATICS

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**"

Note the use of the indirection operator (*) to access the data found at the address stored in **px**.

COLLEGE OF COMPUTING AND INFORMATICS

…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"

COLLEGE OF COMPUTING AND INFORMATICS

…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`)"

COLLEGE OF COMPUTING AND INFORMATICS

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

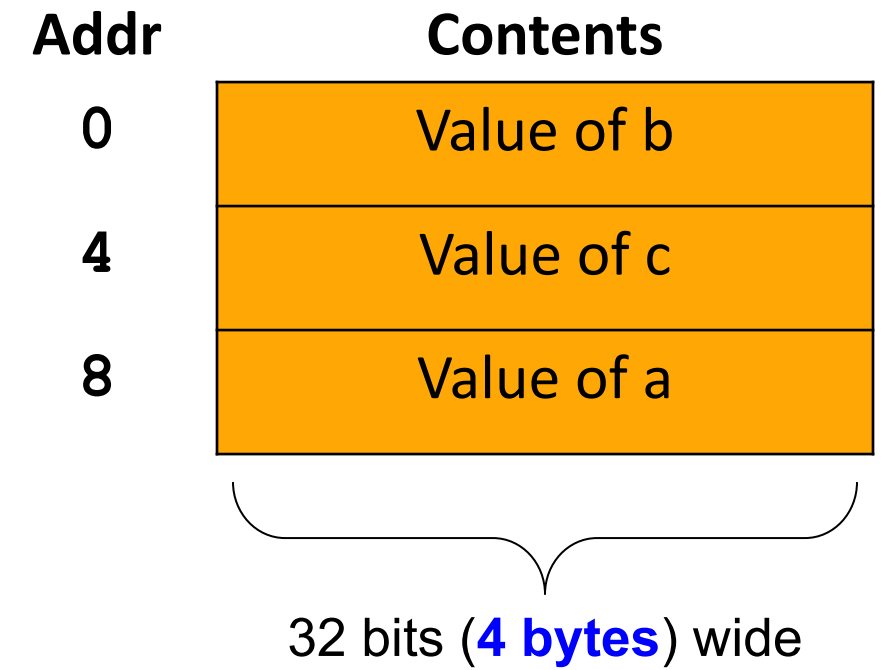
"Pseudo-Assembler" code

```
load int at address 0 into reg1
load int at address 4 into reg2
add reg1 to reg2
store reg2 into address 8
```


COLLEGE OF COMPUTING AND INFORMATICS

Variables Stored in Memory

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



COLLEGE OF COMPUTING AND INFORMATICS

Pointers Refer to Memory Also

A C expression, **with** pointers

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

Symbol Table

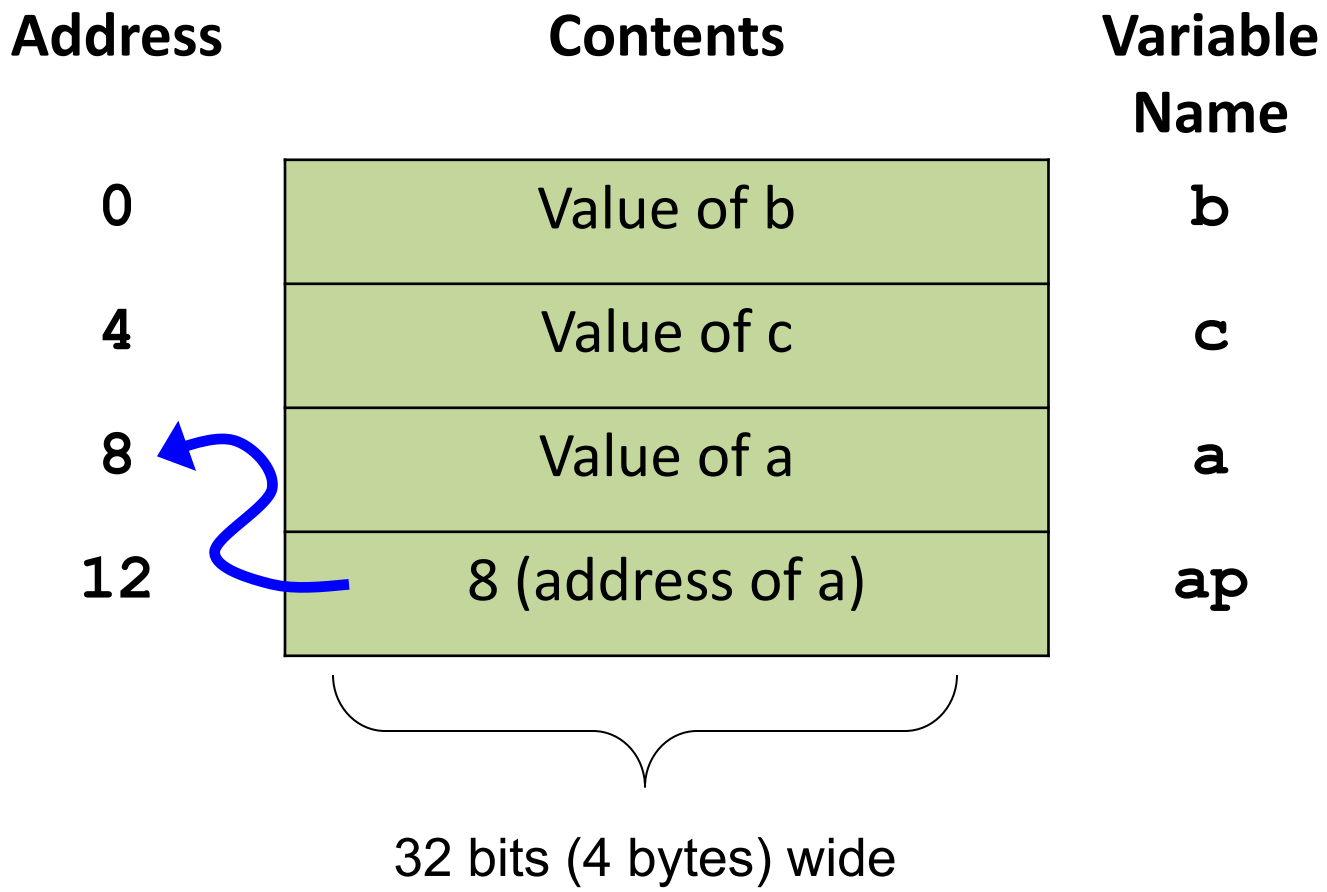
Memory Address	Variable
0	b
4	c
8	a
12	ap

"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
```


COLLEGE OF COMPUTING AND INFORMATICS

Pointers Refer… (cont'd)



COLLEGE OF COMPUTING AND INFORMATICS

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);
```

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*).

Result of execution:

```
a = 35
&a = 3221224568
ap = 3221224568
*ap = 35
```

???

COLLEGE OF COMPUTING AND INFORMATICS

Data Types

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

COLLEGE OF COMPUTING AND INFORMATICS

Pointers and Functions

Passing variables by reference

COLLEGE OF COMPUTING AND INFORMATICS

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

COLLEGE OF COMPUTING AND INFORMATICS

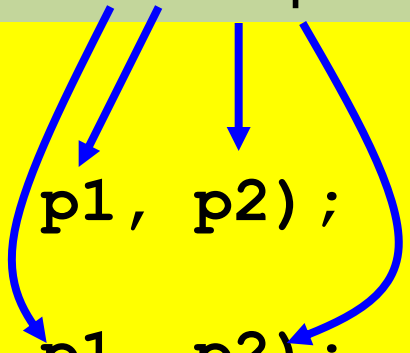
…as Arguments (cont'd)

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

(see **arguments.c** in *Code Samples and Demonstrations* in *Canvas*)

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);
```



COLLEGE OF COMPUTING AND INFORMATICS

…as Arguments (cont'd)

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

COLLEGE OF COMPUTING AND INFORMATICS

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`)?

COLLEGE OF COMPUTING AND INFORMATICS

…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*).

COLLEGE OF COMPUTING AND INFORMATICS

…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 (in Code samples and Demonstrations* in *Canvas)*, execute it and examine the output.

COLLEGE OF COMPUTING AND INFORMATICS

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.

COLLEGE OF COMPUTING AND INFORMATICS

References

- S. J. Matthews, T. Newhall and K. C. Webb, *Dive into Systems*, Version 1.2. Free online textbook, available at: https://diveintosystems.org/book/

- 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.

COLLEGE OF COMPUTING AND INFORMATICS