Debugging

CSC 230 : C and Software Tools

NC State Department of Computer Science

Topics For Today

- Meet assert()
- Meet valgrind (your new best friend)
- Meet cppcheck (a new pretty-good friend)
- Bases in C

assert() in C

- We can build tests right into our source code
- Like Java, C has support for assert()
  - It's provided via the preprocessor:
    #include <assert.h>
- Usage:

    `assert(` *some-condition-you-expect-to-be-true* `);`

- We can use this to build sanity-checks into our code

```
#include <assert.h>

int f( int a, int b )
{
   assert( a != 0 && b > 0 );
   …;
}
```

Planning for Failure

- You can use assert() to comment on your assumptions as write.

```
assert( i < length );
…
assert( x >= 0 && y >= 0 );
…
assert( ptr != NULL );
```

- But, these are comments with teeth!
- They will terminate your program if they are violated.

```
Assertion failed: (j >= 0 && j + 1 < len),\
function sortList, file bubble3b.c, line 50.
```

Assert as Executable Comments

- We had a good example of assert in our resizable array

```
if ( len >= capacity ) {
  capacity *= 2;
  list = (int *)realloc( list, capacity * sizeof( int ) );
}

assert( len < capacity );

list[ len++ ] = val;
```

Grow the array if needed.

It really, really better be big enough now

… because I'm about to use that next sot.

Using and Not Using assert()

- With assert(), we can selectively include bounds checking or other sanity checks in our code.
- Checking assertions cause additional run-time overhead,
  - But, we can use conditional compilation to disable assertions in production code:

```
gcc -DNDEBUG -std=c99 -Wall …
```

Meet your new Best Friend

- Valgrind : an dynamic-analysis error detection tool
- It inserts error-checking instructions as it runs your code.
  - So, there can be considerable overhead, between 10x and 100x
  - Watching what our program does, and complaining about things it doesn't like
- Supports multiple tools, to check for different things.
  - We'll see two of them.

Meet Valgrind memcheck

- The default valgrind tool is memcheck
  - Command-line option: --tool=memcheck
  - Mostly looks for errors with heap-allocated memory
  - Out-of-bounds access to memory
  - Leaked memory
  - Command-line option: --leak-check=full
- Also looks for use of uninitialized static, stack or heap memory

But, this is the default anyway.

Using Valgrind

- Valgrind can tell you more if you include symbol information in your executable:

```
gcc -g -std=c99 –Wall program.c -o program
```

- In general, you run valgrind like this:

```
valgrind valgrind-options ./program program-options
```

- For example, you could check for memory errors and leaks with:

```
valgrind --tool=memcheck --leak-check=full ./program
```

A Bad Program

```
int staticArray[ 10 ];

int main()
{
  int stackArray[ 10 ];

  int *heapArray = (int *)malloc( 10 * sizeof( int ) );

  for ( int i = 0; i <= 10; i++ )
    heapArray[ i ] = 0;

  for ( int i = 0; i <= 10; i++ )
    staticArray[ i ] = 0;

  for ( int i = 0; i <= 10; i++ )
    stackArray[ i ] = 0;
```

Three regions of memory.

There's your problem right there.

Using Valgrind memcheck

- valgrind may generate a **lot** of output

```
$ valgrind --tool=memcheck --leak-check=full ./bounds
==4079== Memcheck, a memory error detector
==4079== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward …
==4079== Using Valgrind-3.12.0 and LibVEX; rerun with -h for …
==4079== Command: ./bounds
==4079==
==4079== Invalid write of size 4
==4079==    at 0x4005B0: main (bounds.c:22)
==4079==  Address 0x51f9068 is 0 bytes after a block of size 40 …
==4079==    at 0x4C29BE3: malloc (vg_replace_malloc.c:299)
==4079==    by 0x40058E: main (bounds.c:17)
<more>
```

Here's what went wrong.

Here's where it went wrong.

Here's what you tried to access.

Here's where you allocated the block.

Meet Valgrind sgcheck

- Memcheck doesn't try to detect out-of-bounds access to stack or static arrays
- There's an experimental tool that tries to do that.
  - Command-line option: --tool=exp-sgcheck
  - It's for stack and global bounds checking
  - It applies some heuristic rules for detecting errors … it may miss some.
- You can enable sgcheck or memcheck, not both
  - But, you can run one then the other.

Using Valgrind sgcheck

```
$ valgrind --tool=exp-sgcheck ./bounds
==4925== … (greetings from valgrind) …
==4925== Command: ./bounds
==4925==
==4925== Invalid write of size 4
==4925==    at 0x4005CE: main (bounds.c:25)
==4925==  Address 0x601088 expected vs actual:
==4925==  Expected: global array "staticArray" of size 40 in object …
==4925==  Actual:   unknown
==4925==  Actual:   is 0 after Expected
==4925==
==4925== Invalid write of size 4
==4925==    at 0x4005F1: main (bounds.c:28)
==4925==  Address 0xffefff08 expected vs actual:
==4925==  Expected: stack array "stackArray" of size 40 in this frame
==4925==  Actual:   unknown
==4925==  Actual:   is 0 after Expected
<a little more>
```

Static array out-of-bounds.

Stack array out-of-bounds.

Understanding Valgrind

- Understanding valgrind output will help you find bugs in your code.
  - Even bugs you don't see when you run the program.
- Valgrind produces lots of output
  - But, it's worth you time to look through to see if it finds any errors.

Invalid Read/Write

- Valgrind will complain if you access memory beyond what you allocated.

```
int *list = (int *)malloc( 10 * sizeof( int ) );

int a = list[ 10 ];

list[ -1 ] =  a + 1;
```

```
==21537== Invalid read of size 4
==21537==    at 0x400522: main (invalid.c:13)
==21537==  Address 0x4c2706c is 0 bytes after a block of size 40 alloc'd
   ...
==21537== Invalid write of size 4
==21537==    at 0x400535: main (invalid.c:16)
==21537==  Address 0x4c2703c is 4 bytes before a block of size 40 alloc'd
```

Uninitialized Memory

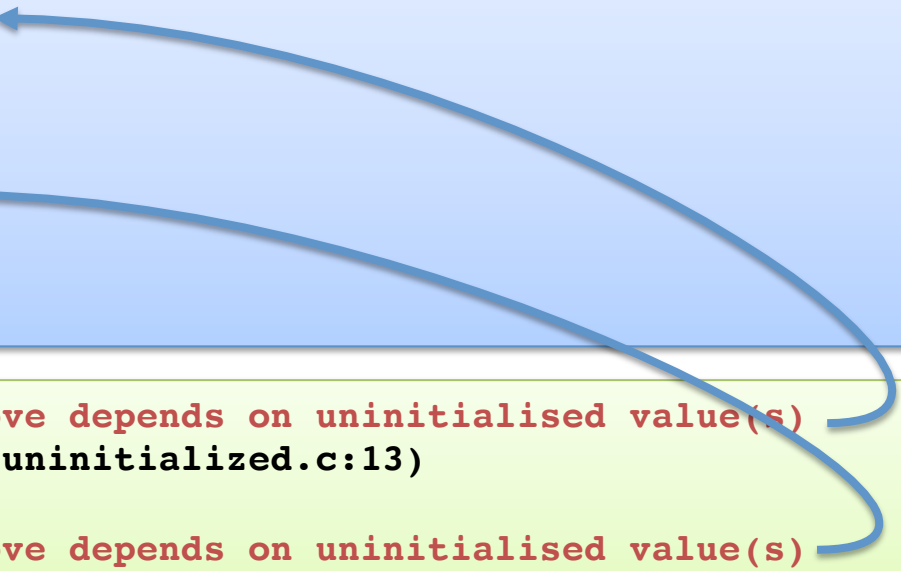
- It will notice if you make choices based on uninitialized memory.

```
int *list = (int *)malloc( 10 * sizeof( int ) );

if ( list[ 0 ] == 0 )
  ...;

int val = list[ 1 ];
if ( val )
  ...;
```

```
==3155== Conditional jump or move depends on uninitialised value(s)
==3155==    at 0x400572: main (uninitialized.c:13)
  ...
==3155== Conditional jump or move depends on uninitialised value(s)
==3155==    at 0x40058F: main (uninitialized.c:19)
```



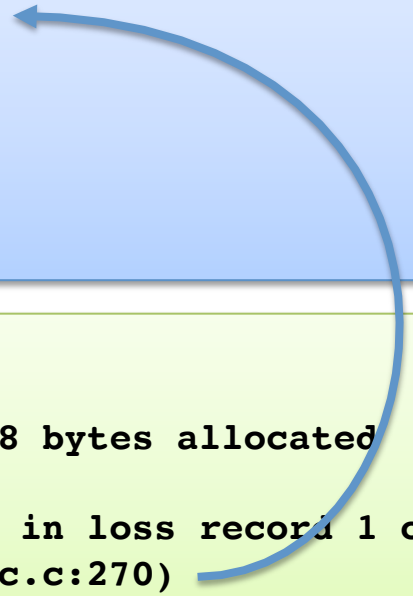
Leaks

- Valgrind will notice if you don't free your memory.

```
int main()
{
  char *buffer = (char *)malloc( 100 );
  FILE *fp = fopen( "abc.txt", "r" );

  return 0;
}
```

```
==3215== HEAP SUMMARY:
==3215==     in use at exit: 668 bytes in 2 blocks
==3215==   total heap usage: 2 allocs, 0 frees, 668 bytes allocated
==3215==
==3215== 100 bytes in 1 blocks are definitely lost in loss record 1 of 2
==3215==    at 0x4A06A2E: malloc (vg_replace_malloc.c:270)
==3215==    by 0x4005D5: main (leak.c:11)
```



File Leaks

- With the right options, valgrind will report file leaks also.

```
int main()
{
  char *buffer = (char *)malloc( 100 );
  FILE *fp = fopen( "abc.txt", "r" );

  return 0;
}
```

```
valgrind --track-fds=yes ./program
```

```
==3250== FILE DESCRIPTORS: 4 open at exit.
==3250== Open file descriptor 3: abc.txt
==3250==    at 0x37F08DB3B0: __open_nocancel (in /lib64/libc-2.12.so)
==3250==    by 0x37F0872AEE: _IO_file_fopen@@GLIBC_2.2.5 (in ...)
==3250==    by 0x37F0866F25: __fopen_internal (in /lib64/libc-2.12.so)
==3250==    by 0x4005EE: main (leak.c:12)
```

Static Analysis

- Valgrind is a dynamic analysis tool
  - It depends on running your program … and seeing if anything goes wrong.
- A static analysis tool depends on looking at your code
  - … and seeing if anything looks suspicious.
- We'll try out cppcheck

Remember Me?

```
int staticArray[ 10 ];

int main()
{
  int stackArray[ 10 ];

  int *heapArray = (int *)malloc( 10 * sizeof( int ) );

  for ( int i = 0; i <= 10; i++ )
    heapArray[ i ] = 0;

  for ( int i = 0; i <= 10; i++ )
    staticArray[ i ] = 0;

  for ( int i = 0; i <= 10; i++ )
    stackArray[ i ] = 0;
```

Three regions of memory.

Out-of-bounds errors.

Using cppcheck

- We run cppcheck on the source file

```
$ cppcheck bounds.c
Checking bounds.c...
[bounds.c:25]: (error) Array 'staticArray[10]' accessed at index 10,
which is out of bounds.
[bounds.c:28]: (error) Array 'stackArray[10]' accessed at index 10,
which is out of bounds.
[bounds.c:22]: (error) Array 'heapArray[10]' accessed at index 10,
which is out of bounds.
```

It found all three errors.

A Little Obfuscation

```
void zeroFill( int *a )
{
  for ( int i = 0; i <= 10; i++ )
    a[ i ] = 0;
}

int staticArray[ 10 ];

int main()
{
  int stackArray[ 10 ];

  int *heapArray = (int *)malloc( 10 * sizeof( int ) );

  zeroFill( heapArray );
  zeroFill( staticArray );
  zeroFill( stackArray );
```

The out-of-bounds access is now in this function.

Missing Errors with cppcheck

- Static analysis can't possibly detect every error

```
$ cppcheck bounds2.c
Checking bounds2.c...
```

Oops, didn't see the errors this time.

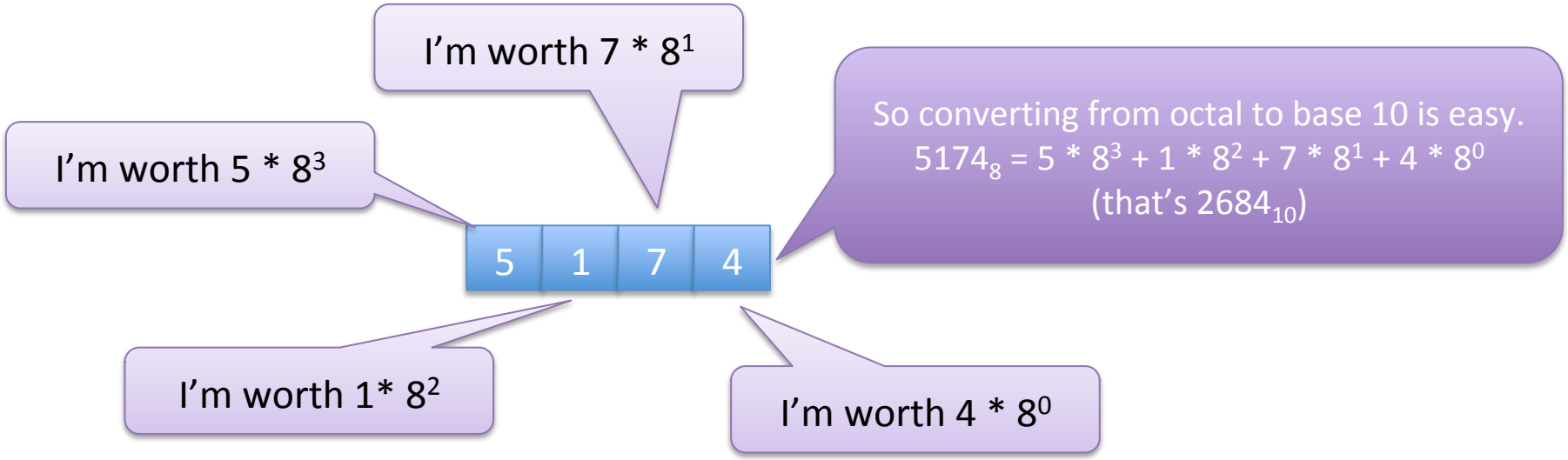
- It's good to have access to a combination of tools.
- In class, we'll compare our static- and execution-time tools.

Covering Your Bases

- The C language lets/makes us work in different bases
- We need to be good at working with these bases and switching among them
  (well, at least we need to be able to do it)
- Octal and hexadecimal are like compact representations of binary
  - They expose the binary representation, without taking a huge amount of space
  - Octal uses a symbol for each group of 3 bits, Hexadecimal uses a symbol for groups of 4

Meet Octal

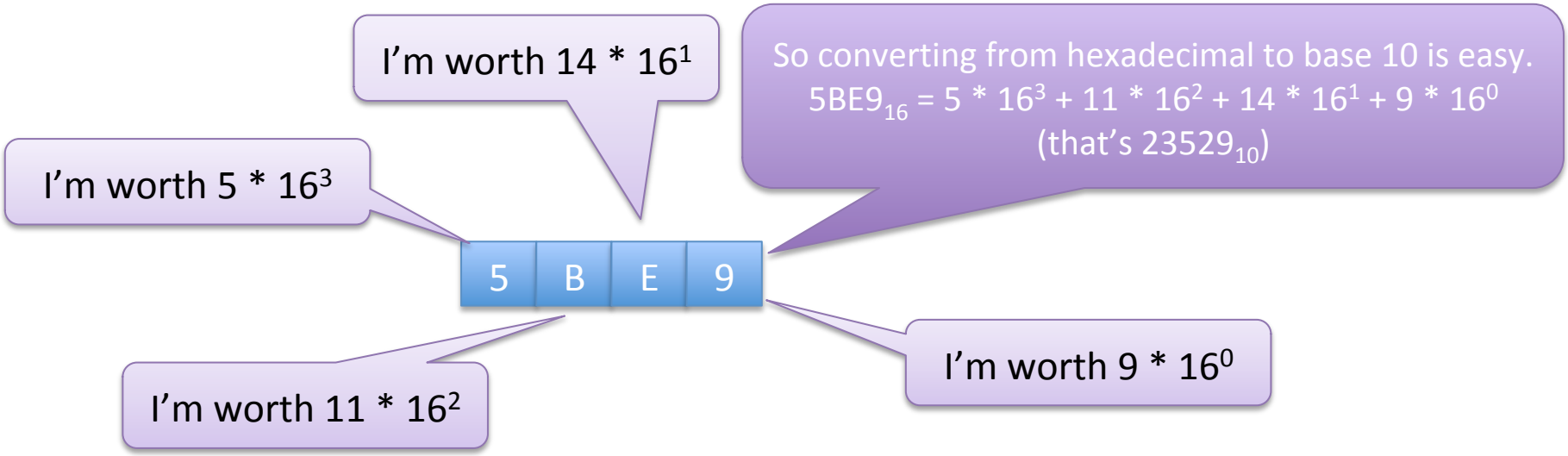
- Octal is a base-8 representation
  - Each octal digit represents 3 bits
  - We need symbols 0 – 7
  - We use notation 5174_8 to show that a number is octal



Meet Hexadecimal

- Hexadecimal is a base-16 representation
  - Each hexadecimal digit represents 4 bits
  - We need 16 different symbols
  - Digits 0 – 9 have their usual value
  - For the rest, we just start using letters

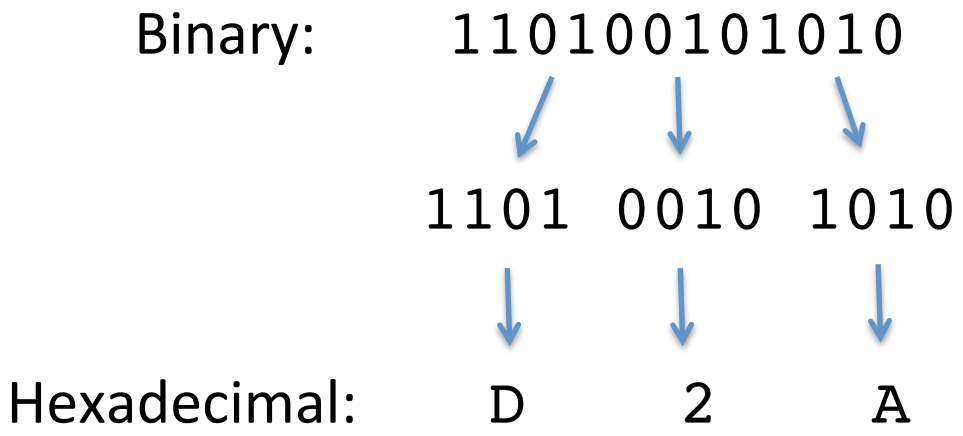
A is worth 10	B is worth 11
C is worth 12	D is worth 13
E is worth 14	F is worth 15



Covering Your Bases

- Binary to Hexadecimal
  - Break into groups of 4 bits
  - Convert each group

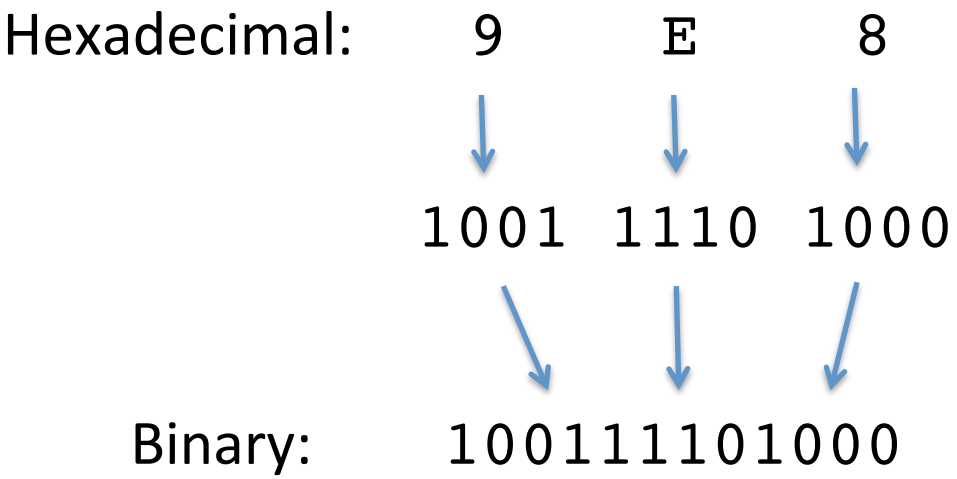
0000 → 0	0001 → 1	0010 → 2	0011 → 3
0100 → 4	0101 → 5	0110 → 6	0111 → 7
1000 → 8	1001 → 9	1010 → A	1011 → B
1100 → C	1101 → D	1110 → E	1111 → F



Covering Your Bases

- Hexadecimal back to binary
  - Replace each symbol with its equivalent bit pattern

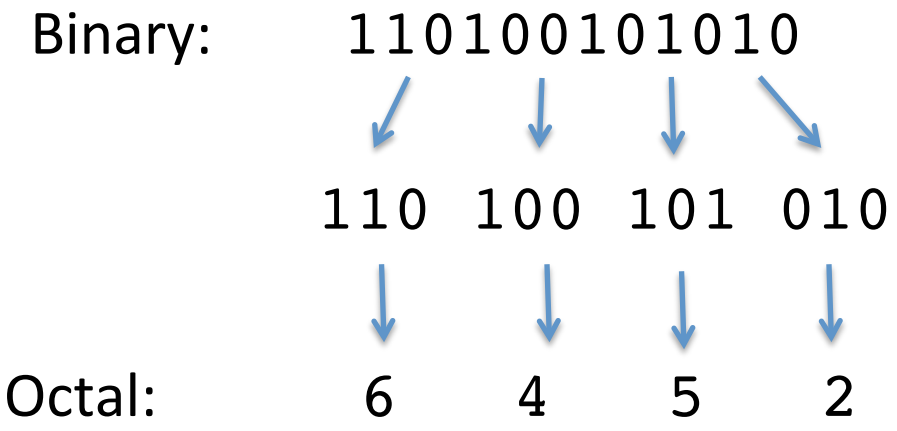
0 → 0000	1 → 0001	2 → 0010	3 → 0011
4 → 0100	5 → 0101	6 → 0110	7 → 0111
8 → 1000	9 → 1001	A → 1010	B → 1011
C → 1100	D → 1101	E → 1110	F → 1111



Covering Your Bases

- Binary to Octal
  - Break into groups of 3 bits
  - Convert each group

000 → 0	001 → 1	010 → 2	011 → 3
100 → 4	101 → 5	110 → 6	111 → 7



Decimal to Hexadecimal

- Imagine we want to convert 2687 from decimal to hexadecimal

- Mod by 16 gives us the low-order digit
  - Why? Well, all the higher-order digits are worth some multiple of 16.

Decimal to Hexadecimal

- Dividing by 16 discards the low-order hexadecimal digit.
- And, it shifts the remaining digits down by one position.

- Now, we can repeat to recover the next-lowest-order digit.

Decimal to Hexadecimal

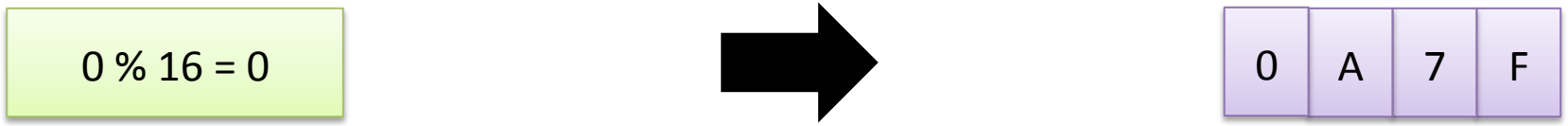
- Keep going to recover the rest of the number.
- Divide to shift to the next hexadecimal digit.

- Mod to obtain the next digit.

Decimal to Hexadecimal

- Divide to shift to the next hexadecimal digit.

- I guess we're done.
  - Any other high-order hexadecimal digits we computed would be zeros anyway.



Decimal to Hexadecimal

- This sounds like an algorithm.
  - Given some value *v*
  - While *v* is non-zero
    - Compute *v* % 16, the value of the next lowest-ordered digit
    - Let *v* = *v* / 16 to shift to the next lowest-ordered digit

- I bet this would work for other bases also … binary … or even octal

Decimal to Octal

- In fact, we can use the same technique to convert decimal to octal
  - Given some value *v*
  - While *v* is non-zero
    - Compute *v* % 8, the value of the next lowest-ordered digit
    - Let *v* = *v* / 8 to shift to the next lowest-ordered digit

- Why do computer scientists confuse Halloween and Christmas?

  (Because, to a Computer Scientist, Oct 31 and Dec 25 are the same thing.)

Octal and Hexadecimal in C

- In C, you can give an octal constant by starting it with a 0 (zero).

```
int a = 0123;
```

- Or, a hexadecimal constant by starting it with 0x (zero-x)

```
int b = 0xABC;
```

Either capital or lower-case is OK.

- Java also lets us do this, but maybe you never used it.

Octal and Hexadecimal I/O

- We can print an octal int value using %o

```
printf( "%o\n", a );
```

- Or read an octal int value with %o

```
scanf( "%o", &a );
```

- For printing in hexadecimal, we have %x

```
printf( "%x\n", b );
```

- And it also works for reading values.

```
scanf( "%x", &b );
```

Meet bc

- The program, bc, is a handy little shell command.
- It can perform arithmetic to arbitrary precision
  - Need to know 394284920834342849823940382904839043849403284942834984932489023489408392 * 882838384557893929830989409320849028230? bc can tell you.
- But, really, it's most useful for doing math in your chosen base

```
$ bc
obase=2          # set the output base
ibase=16         # set the input base
3F9 * B2A
10110001011001110110 10
```

It's a good idea to set your obase first.