

CSC 374/407: Computer Systems II

Lecture 1

Joseph Phillips
De Paul University

2014 July 18

Copyright © 2011 Joseph Phillips
All rights reserved

Reading

- ♦ Bryant & O'Hallaron “*Computer Systems, 2nd Ed.*”
 - ♦ Chapter 5.1-5.6, 5.10-5.15
- ♦ Hoover “*System Programming*”
 - ♦ Chapter 1

Topics

- Review: Pointers and objects
- About machine-independent optimization
- Using registers instead of RAM
- Code motion
- Common expression computation
- Reduction in strength
- Limitations
- Profiling
- “*So, how do I actually program knowing this?*”

But first . . let's learn some C:

A little C (Output)

- ◆ Output in C with `printf()` (“print-formatted”)
 - ◆ `printf("template", expr1, ... exprn)`
- ◆ Constant formatting:
 - ◆ `printf("\tI just print \"hello\".\n");`
 - ◆ What do these mean? `\t` `\n`
- ◆ Substitution formatting:
 - ◆ `int i=1; printf("%d %d %d Go!\n", 3, 1+1, i);`
 - ◆ `%d` = decimal integer
 - ◆ `%x %X` = hexadecimal integer
 - ◆ `%c` = single char
 - ◆ `%s` = C-string (*i.e.* pointer to char: `char*`)
 - ◆ `%f %g` = double or floating point
 - ◆ `%p` = An address (*e.g.* a pointer's value)

A little more C (Input):

- ♦ ***Almost always*** should get any input as string
 - ♦ Convert string to integer or float

```
#include <stdio.h>

#define LINE_LEN 10

. . .
char line[LINE_LEN];
printf("Please enter a number: ");
fgets(line, LINE_LEN, stdin);
int i = strtol(line, NULL, 10);
float f = strtod(line, NULL);
printf("i = %d, f = %g\n", i, f);
```

About `strtol()` and `strtod()`

`strtol`
(`const char* text`,
 `char** endPtr`,
 `int base`)
(STRing TO Long)

`strtod`
(`const char* text`,
 `char** endPtr`)
(STRing TO Double)

- ◆ **text**: pointer to text to convert
- ◆ **endPtr**: address of pointer to receive just beyond last char converted (or **NULL** if you don't care)
- ◆ **base**: Base (2-36) to use. Or 0, in which case it uses the same rules as C int constants.

strtol() example

```
#include<stdlib.h>
#include<stdio.h>
#define TEXT_LEN    64
int  main ()
{ char  text[TEXT_LEN];
  char* cPtr;
  while (1)
  {
    printf("Please enter a number: ");
    fgets(text,TEXT_LEN,stdin);
    int i = strtol(text,&cPtr,0); //We're using base 0
    if  (cPtr == text)
      printf("Phooey!\n");
    else
      printf("dec:\t%d\n"
             "hex:\t%X\n",i,i);
  }
  return(EXIT_SUCCESS);
}
```


strtol () example (cont'd)

```
$ ./strtol
```

```
Please enter a number: 20      // Ordinary decimal
```

```
dec: 20
```

```
hex: 14
```

```
Please enter a number: 020    // Leading 0 => octal
```

```
dec: 16
```

```
hex: 10
```

```
Please enter a number: 0x20   // Leading 0x => hex
```

```
dec: 32
```

```
hex: 20
```

```
Please enter a number: twenty // No digit => ERROR!
```

```
Phooey!
```

```
Please enter a number:
```

You *knew* this was coming: addresses and pointers! (1)

- ◆ What is an address?
 - ◆ It is where in memory some object starts.
 - ◆ All objects in memory have unique starting addresses
 - ◆ In general you do **not** specify addresses, the OS/compiler/running system specify them for you.
- ◆ Getting addresses:
 - ◆ `SomeType var;`
`. . .`
`&var` // Returns address of var
 - ◆ `SomeType array[ARRAY_LEN];`
`. . .`
`array` // Returns addr of array[0]
`&array[0]` // Same addr as 'array'
 - ◆ `malloc(16);` // Returns addr of 16 bytes on heap
 - ◆ `"string const"` // Returns addr where
// "string const" starts

You *knew* this was coming: addresses and pointers! (2)

- ◆ Pointers are variables that store addresses
 - ◆ In general, they can store the address of (ie. “point to”) which ever object they want

- ◆ Declaring pointers: *Type* typePtr*

`int* intPtr;`

`char* charPtr;`

`SomeType* someTypePtr;`

You knew this was coming: addresses and pointers! (3)

◆ Putting it together:

```
int          i          = 10 ;  
int*         intPtr1 = &i ;  
int*         intPtr2 = intPtr1 ;  
const char*  charPtr = "string const" ;  
char         buffer[100] ;
```

```
charPtr = buffer ;  
charPtr = (char*)malloc(10) ;
```

```
free(buffer) ;
```

YOUR TURN!

- Consider `sscanf()`:

- `int sscanf`

- `(const char* source, int i, j, n;
const char* format
<, addr1>
<, addr2>
...)`

- Reads values from `source` into `addr1`, `addr2`, using `printf`-like specifiers in `format`.
- Returns num. items read

- Which one sets `i=10`, `j=20`, and `n=2`?

a) `n = int sscanf("10 20", "%d %d", *i, *j)`

b) `n = int sscanf("10 20", "%d %d", &i, &j)`

c) Neither

Dereferencing: 1

- ◆ Dereferencing means “*follow the pointer to the object*”:
- ◆ Done with **objectPtr*
- ◆ Important! Two different stars! (*)

```
// * after type declares ptr var  
int i = 10;  
int* intPtr = &i;
```

```
// * before ptr var dereferences  
printf("i=%d\n", *intPtr);
```

Your turn!

- ♦ Write a program with two variables:
 - ♦ Integer `i`.
 - ♦ Pointer to `i` called `iPtr`.
- ♦ Give the program a for loop that counts from 0 to 9 using the value stored in `i` but that never refers to `i` **except** to initialize `iPtr`.

Dereferencing: 2

- Use arrow (→) to access struct or class member:

```
class XYCoord
{int x; int y;
 public:
  XYCoord(int nX,int nY) {x=nX; y=nY; }
  int getX() const { return(x); }
  int getY() const { return(y); }
};

. . .
XYCoord* pPtr = new XYCoord(1,2);
printf( "%d,%d\n",
        pPtr->getX(),pPtr->getY() );
delete(pPtr);
```


Your turn!

- Write a method `print()` for the class on the previous slide
- Call your method using the pointer `ptr`
- These are equivalent:

```
// Do method() of pointed-to obj  
ptr->method();
```

```
// Get pointed-to obj then do method()  
(*ptr).method();
```

Arrays and pointers

- ♦ Arrays are actually addresses:
 - ♦ Address of their first item
 - ♦ Can be used:
 - ♦ to assign values to other pointers
 - ♦ to test their values against other pointers
 - ♦ Their position cannot be changed by ++, *etc.*
- ♦ Pointers can also be accessed like arrays:
- ♦ Can dereference with:
 - ♦ `*ptr` or `ptr[0]`
 - ♦ `ptr->method()` or `(*ptr).method()` or `ptr[0].method()`

Your turn!

```
// Example
int  intArray[5] = {10,20,30,40,50};
int* intPtr      = intArray;

// What will this print?
printf("%p %p\n",intArray,intPtr);

(*intPtr) *= 10;

// What will this print?
printf("%d\n",intArray[0]);
```

Const pointers

- ◆ const SomeType* ptr;
- ◆ Same promise as const references:
 - ◆ “I, the programmer, promise not to change the pointed to object”
 - ◆ Compiler will yell at you if:
 - ◆ You break that promise:

```
const int* cIntPtr = &i;  
(*cIntPtr)++; // Illegal!
```
 - ◆ You pass it to something that does not maintain that promise:

```
const int* cIntPtr = &i;  
int*      IntPtr  = cIntPtr;    // Illegal!
```
 - ◆ You may, however, point at different objects:

```
const int* cIntPtr = &i;  
cIntPtr    = &j;    // Okay
```

Your turn!

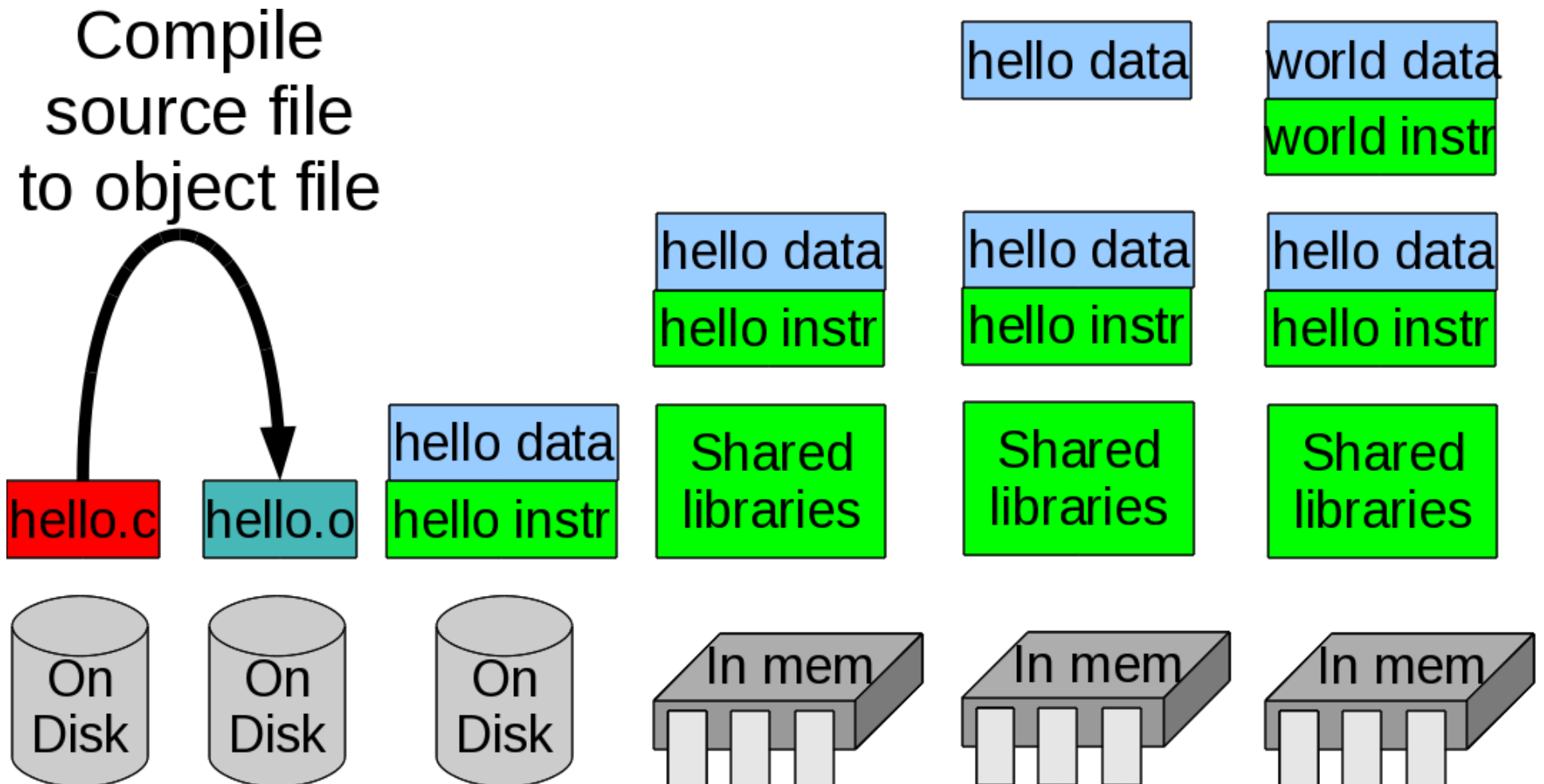
- ◆ Happy compiler :) or sad compiler :(?

```
void compilerPisserOffer(const int* ciPtr)
{
    int* iPtr = ciPtr;    // Happy or sad?
    int array[5] = {1,2,3,4,5};

    ciPtr = array; // Happy or sad?
    ciPtr++;       // Happy or sad?
    (*ciPtr)++;    // Happy or sad?
}
```

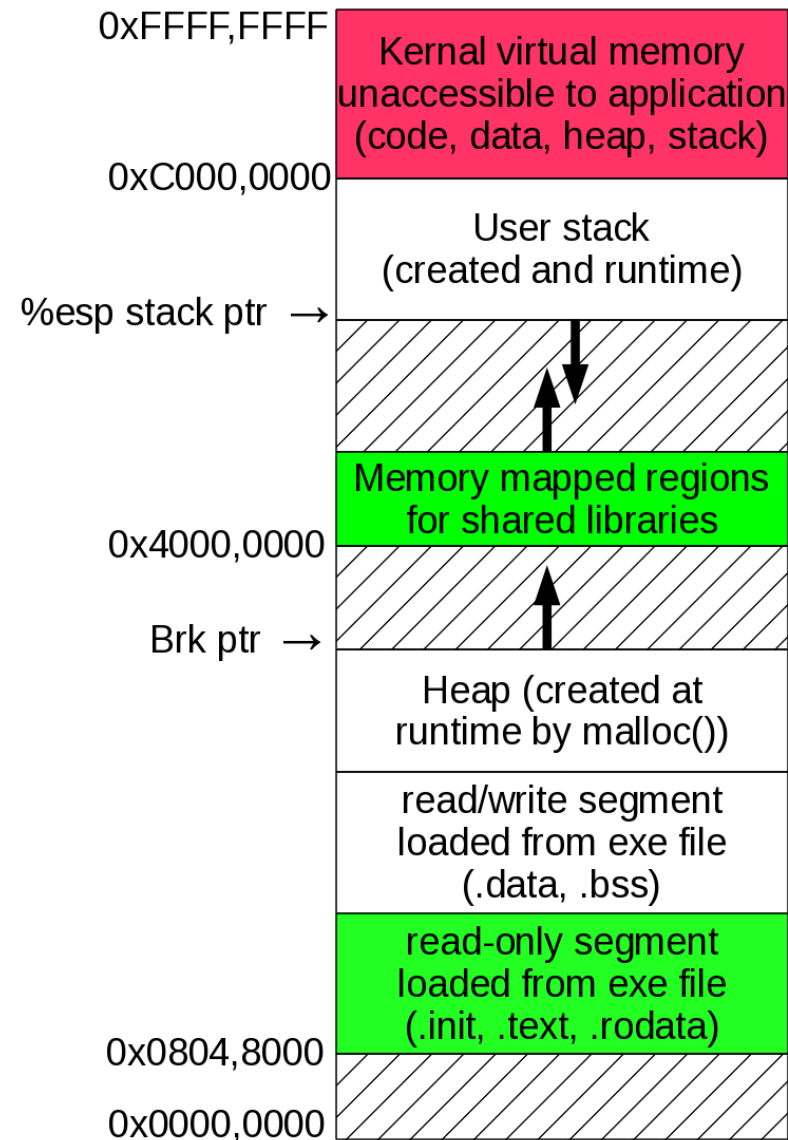
Today's topic (in time)

How to compile to create efficient programs:



Today's topic (in space)

- Efficient code for a program to run



Speed it up!

- ◆ Q: Why do we just love computers? (They don't have any common sense after all)
- ◆ A1: Because they don't get bored
- ◆ A2: Because they are accurate
- ◆ A3: Because they're ***fast*** baby!

Speed it up! (2)

- ♦ What goes into making an algorithm fast?
 1. Algorithm and data-structure choice
 - ♦ Don't do $O(N)$ linear search in a linked list, *Ninny!*
 - ♦ Do $O(\lg N)$ binary search in a balanced tree instead
 - ♦ This is important, it's why we harass you with a basic and an advanced course in data structures and algorithms
 2. Implementing the algorithm and data structure
 - ♦ The compiler has to change it into assembly language
 - ♦ ***That's the aim of this lecture!***

Speed it up! (3)

- ◆ We will study machine independent optimizations
 - ◆ Make sense to do no matter what your CPU
- ◆ What can we do?
 1. Use our registers efficiently
 - ◆ It takes time to load/store data
 2. Re-order code
 - ◆ Don't do the same thing multiple times if don't have to
 3. Get rid of obviously inefficient things

Speed it up! (4)

- ◆ Watch out! Good optimizations:

1. Don't change the programmer's data-structure behind his/her back (*Why not?*)
2. Don't change the programmer's algorithm behind his/her back (*Why not?*)
3. Watch out for memory aliasing:

```
int i = 10;  
int* iPtr = &i;  
(*iPtr)++;  
printf("i is still 10, right? %d\n", i);
```

4. Don't assume too much about what other fncs do

Speed it up! (5)

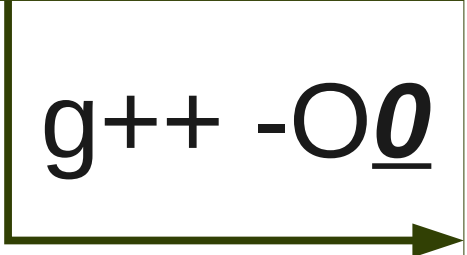
- ◆ Our approach:
 1. Only make code equivalent to what the programmer wrote (even “optimizing” their bugs)
 - ◆ When the compiler spots something that looks like a bug what should it do?
 2. Optimize functions individually
 3. Don't make any assumptions about inputs
- ◆ Four main tools:
 1. Store in **registers**, not RAM.
 2. **Code motion**: move code so it isn't being done so many times (e.g. loops)
 3. Compute **common expressions** once
 4. **Reduction in strength**: Use a cheaper operator.

Using Registers

Without optimization most variables are kept in RAM (e.g. the stack)

```
int counter (int limit)
{ int sum= 0;
  int i;
  for (i = 0; i <= limit; i++)
    sum += i;
  return(sum);
}
```

g++ -O0



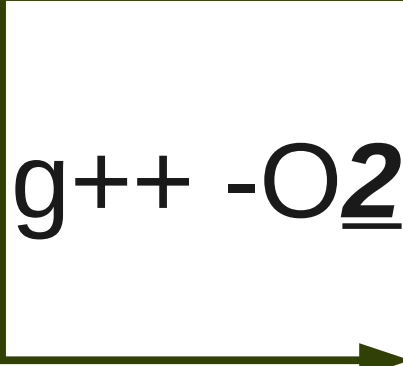
```
<+0>:  push    %ebp
<+1>:  mov     %esp,%ebp
<+3>:  sub     $0x10,%esp
<+6>:  movl    $0x0,-0x4(%ebp)  sum=0
<+13>: movl    $0x0,-0x8(%ebp)  i=0
<+20>: jmp     <counter+32>
<+22>: mov     -0x8(%ebp),%eax
<+25>: add     %eax,-0x4(%ebp)  sum+=i
<+28>: addl    $0x1,-0x8(%ebp)  i++
<+32>: mov     -0x8(%ebp),%eax
<+35>: cmp     0x8(%ebp),%eax  i<=limit?
<+38>: jle     <counter+22>
<+40>: mov     -0x4(%ebp),%eax
<+43>: leave
<+44>: ret
```

Using Registers (2)

```
int counter (int limit)
{ int sum= 0;
  int i;
  for (i = 0; i <= limit; i++)
    sum += i;
  return(sum);
}
```

With optimization some variables are kept in registers

g++ -O2



```
<+0>: mov    0x4(%esp),%ecx ecx = limit
<+4>: xor     %eax,%eax eax=sum=0
<+6>: test    %ecx,%ecx limit==0?
<+8>: js      <counter+25> // jmp to ret
<+10>: xor     %edx,%edx edx=i=0
<+12>: lea     0x0(%esi,%eiz,1),%esi
<+16>: add     %edx,%eax sum+=i
<+18>: add     $0x1,%edx i++
<+21>: cmp     %edx,%ecx i<=limit
<+23>: jge     <counter+16>
<+25>: repz    ret // optimized return
```

Code motion

- Move code so not doing redundant calculations
 - Might introduce temporary variables to hold results

```
for (i = 0; i < n; i++)  
    for (j = 0; j < n; j++)  
        a[i*n + j] = b[j]
```



```
for (i = 0; i < n; i++)  
{  
    int in = i*n;  
    for (j = 0; j < n; j++)  
        a[in + j] = b[j]  
}
```

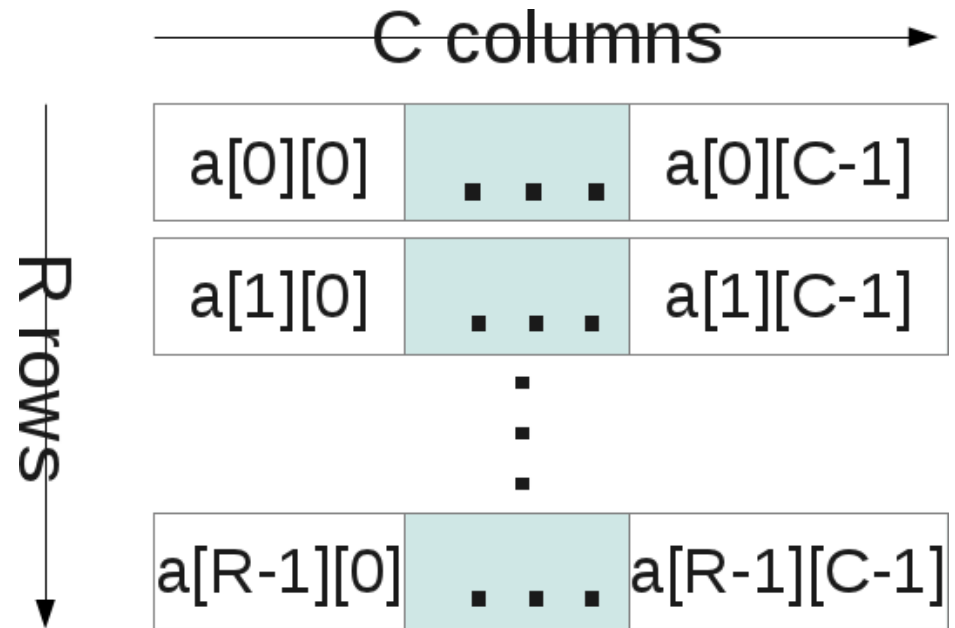
Your turn!

- ◆ Please optimize using code motion:

```
for (i = 0; i < n; i++)  
    for (j = 0; j < n; j++)  
        for (k = 0; k < j; k++)  
            a[i*n + i*j - k] = b[k*7];
```

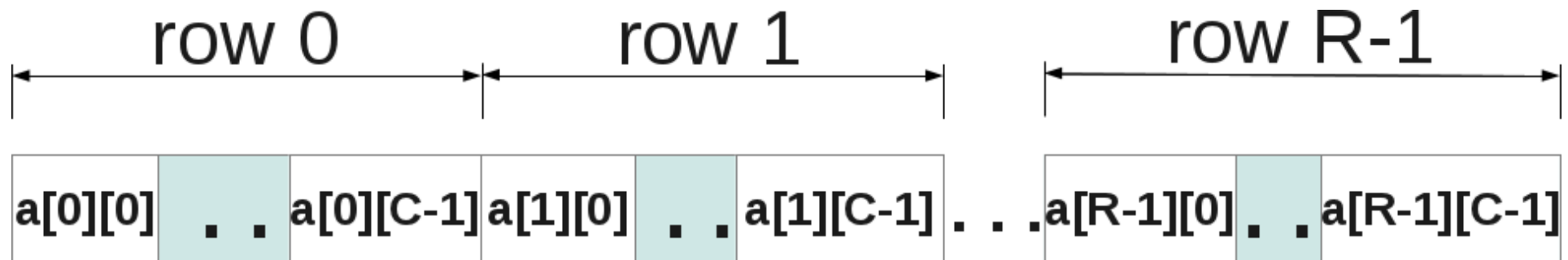

Compute common expressions once

- Our example will use 2-D arrays
- So let's see how 2-D arrays are laid out in memory



Compute common expressions once

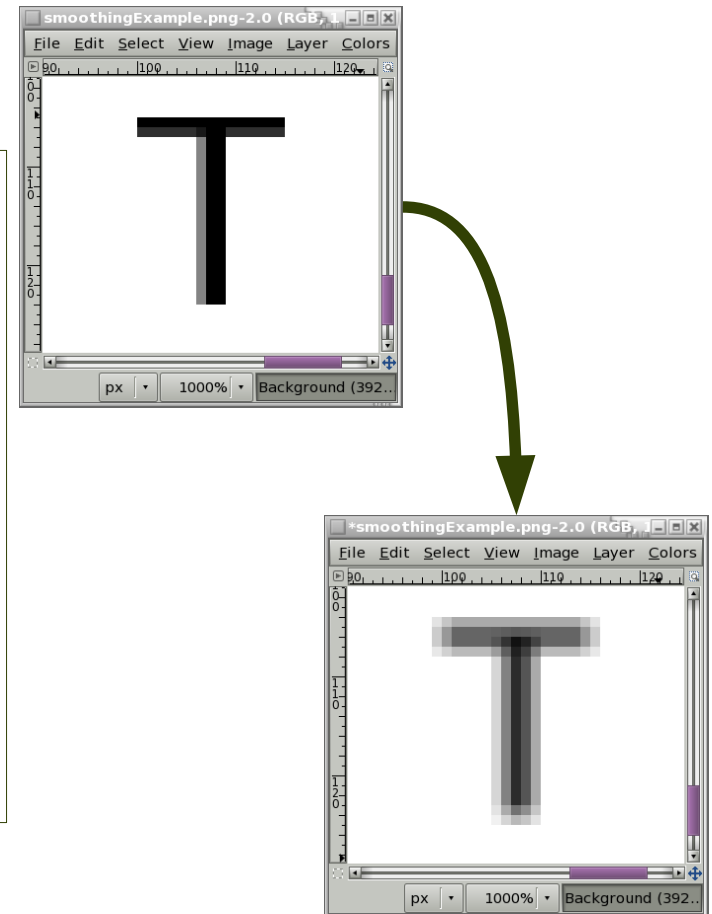
- 2-D arrays are actually laid out in memory as very long 1-D arrays
- $\&\text{array2D}[i][j] == \&\text{array1D}[i * \text{NumCols} + j]$



Compute common expressions once

- Now, consider the following:
 - Actually done in image processing (e.g. blurring)

```
above = array[n*(i-1)+j];  
below = array[n*(i+1)+j];  
left  = array[i*n+j-1];  
right = array[i*n+j+1];  
  
aver  = (above+below+  
         right+left )/4;
```



Compute common expressions once, cont'd

- Don't compute what's common more than once:
 - They all share $i*n+j$ in various guises:

```
inPj    = i*n+j;  
above   = array[inPj - n];  
below   = array[inPj + n];  
left    = array[inPj - 1];  
right   = array[inPj + 1];  
  
aver    = (above+below+right+left )/4;
```

Your turn!

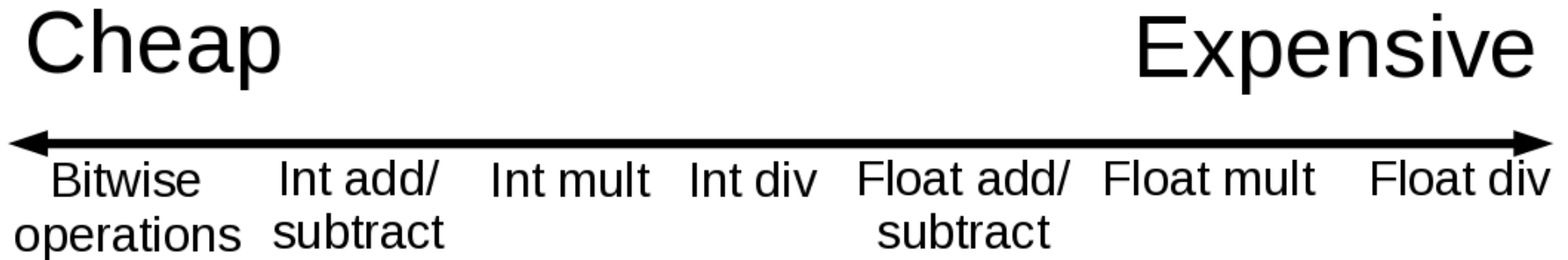
► Optimize:

```
above = array[ (n*n) * (i+1) + j*n      +k ] ;
below = array[ (n*n) * (i-1) + j*n      +k ] ;
left  = array[ i*n*n          +n* (j-1) +k ] ;
right = array[ i*n*n          +n* (j+1) +k ] ;
front = array[ i*n*n          +j*n      +k-1 ] ;
back  = array[ i*n*n          +j*n      +k+1 ] ;

aver = (above+below+
        right+front+
        back+left) / 6 ;
```

Reduction in strength

- ◆ Change from expensive operation to cheaper
- ◆ What's expensive?
 - ◆ Depends on CPU
- ◆ Generally:



Reduction in strength

- ◆ It makes sense to do:
 - ◆ << instead of multiplying by powers of 2
 - ◆ >> instead of dividing by powers of 2
 - ◆ Additions instead of multiplications
- ◆ Not so important now, but “i*10” used to be done as:

```
int result    = i << 2;           // i*4
int result += i;                   // i*5
int result    = result << 1;      // i*10
```

- ◆ Rather do 2 shifts and addition than 1 multiply

Reduction in strength, ex

```
unsigned int num=strtol(line,0,0);  
//call    0x8048360 <strtol@plt>  
//mov     %eax,0xffffffff0(%ebp)
```

```
unsigned int numDiv4 = num / 4;  
//mov     0xffffffff0(%ebp),%eax  
//shr     $0x2,%eax  
//mov     %eax,0xffffffff4(%ebp)
```

```
unsigned int numMul4 = num * 4;  
//mov     0xffffffff0(%ebp),%eax  
//shl     $0x2,%eax  
//mov     %eax,0xffffffff8(%ebp)
```


Reduction in strength, ex 2

Trickier dance if dealing with signed numbers:

```
int num = strtol(line,0,0);  
//call    0x8048360 <strtol@plt>  
//mov     %eax,0xffffffff0(%ebp)
```

```
int numDiv4 = n / 4;  
//mov     0xffffffff0(%ebp),%edx  
//mov     %edx,%eax  
//sar     $0x1f,%eax  
//shr     $0x1e,%eax  
//add     %edx,%eax  
//sar     $0x2,%eax  
//mov     %eax,0xffffffff4(%ebp)
```

Your turn!

Compute the following without using `imult` or `idiv`:

- $i * 3$
- $i * 12$
- $i * 20$
- $i * 31$

Is it practical to compute $i * j$ without `imult`?

Code motion + reduction in strength

- ◆ It makes sense to do:
 - ◆ Repeated adding to a temp var instead of repeated multiplying the same num by 0, 1, 2, 3, *etc.*
 - ◆ Repeated subtracting from temp var instead of repeated multiplying the same num by 9, 8, 7, *etc.*
 - ◆ Uses temporary variable to accumulate sum
 - ◆ **Also**, can use pointers to march thru arrays

Code motion + reduction in strength, cont'd

- CM: Move code out of inner loop(s)
- RiS: Change repeated multiplies to repeated adds

```
for (i = 0; i < n; i++)  
    for (j = 0; j < n; j++)  
        a[i*n + j] = b[j]
```



```
int in = 0;  
for (i = 0; i < n; i++)  
{  
    for (j = 0; j < n; j++)  
        a[in + j] = b[j]  
    in += n;  
}
```

Your turn!

- ◆ ***Remember me?***
 - ◆ Optimize me again, getting rid of as many expensive multiplies as you can . . .

```
for (i = 0; i < n; i++)  
    for (j = 0; j < n; j++)  
        for (k = 0; k < j; k++)  
            a[i*n + i*j - k] = b[k*7];
```

A real world example:

```
#include <stdlib.h>
```

```
int main ()
```

```
{
```

```
int i;
```

```
int j;
```

```
int n = 100;
```

```
int* a = (int*)
```

```
    calloc(n*n,sizeof(int));
```

```
int* b = (int*)
```

```
    calloc( n,sizeof(int));
```

```
    for (i = 0; i < n; i++)  
        for (j = 0; j < n; j++)  
            a[n*i + j] = b[j];
```

```
    free(b);
```

```
    free(a);
```

```
    return(0);
```

```
}
```

Optimize 00 (none)

<main+72>:	movl	\$0x0,0xffffffffe8(%ebp)	i = 0
<main+79>:	jmp	<main+137>	
<main+81>:	movl	\$0x0,0xffffffffec(%ebp)	j = 0
<main+88>:	jmp	<main+125>	
<main+90>:	mov	0xfffffffff0(%ebp),%eax	eax = n
<main+93>:	imul	0xffffffffe8(%ebp),%eax	eax = n*i
<main+97>:	add	0xffffffffec(%ebp),%eax	eax = n*i + j
<main+100>:	shl	\$0x2,%eax	eax = (n*i + j)*4
<main+103>:	mov	%eax,%edx	edx = eax
<main+105>:	add	0xfffffffff4(%ebp),%edx	edx = a+(n*i+j)*4
<main+108>:	mov	0xffffffffec(%ebp),%eax	eax = j
<main+111>:	shl	\$0x2,%eax	eax = j * 4
<main+114>:	add	0xfffffffff8(%ebp),%eax	eax = b + j * 4
<main+117>:	mov	(%eax),%eax	eax = mem[b+j*4]
<main+119>:	mov	%eax,(%edx)	mem[a+(n*i+j)*4]=eax
<main+121>:	addl	\$0x1,0xffffffffec(%ebp)	j++
<main+125>:	mov	0xffffffffec(%ebp),%eax	eax = j
<main+128>:	cmp	0xfffffffff0(%ebp),%eax	j < n
<main+131>:	j1	<main+90>	?
<main+133>:	addl	\$0x1,0xffffffffe8(%ebp)	i++
<main+137>:	mov	0xffffffffe8(%ebp),%eax	eax = i
<main+140>:	cmp	0xfffffffff0(%ebp),%eax	i < n
<main+143>:	j1	<main+81>	?

Optimize O1 (First level)

```
<main+42>:    mov     %eax,0xffffffff(%ebp)  a = calloc(10000,4)
<main+45>:    movl    $0x4,0x4(%esp)        push 4
<main+53>:    movl    $0x64,(%esp)            push 100
<main+60>:    call    0x8048388 <calloc@plt>
<main+65>:    mov     %eax,%ebx                ebx = (b =calloc(100,4))
<main+67>:    mov     0xffffffff(%ebp),%esi     esi = a
<main+70>:    jmp     <main+105>
<main+72>:    mov     0xfffffffffc(%ebx,%edx,4),%eax
                                           eax = mem[b + 4*(j+1) - 4]
<main+76>:    mov     %eax,(%ecx)              *a = eax
<main+78>:    add     $0x1,%edx                (j+1)++
<main+81>:    add     $0x4,%ecx                a++
<main+84>:    cmp     $0x65,%edx                (j+1) != 101
<main+87>:    jne     <main+72>                  ?
<main+89>:    add     $0x190,%esi               esi += 400
<main+95>:    lea     0x9c40(%edi),%eax         eax = a + 40,000
<main+101>:   cmp     %eax,%esi                eax == esi
<main+103>:   je      <main+114>                ?
<main+105>:   mov     %esi,%ecx                ecx = a
<main+107>:   mov     $0x1,%edx                edx = (j+1) = 1
<main+112>:   jmp     <main+72>
<main+114>:   mov     %ebx,(%esp)
```


Limitations

- ♦ Harder for compiler to optimize across function calls
 - ♦ **Even though you would *want* to!**
- ♦ Example:
 - ♦ Is this efficient?

```
void stupidUppercase (char* string)
{
    int i;
    for (i = 0; i < strlen(string); i++)
        string[i] = toupper(string[i]);
}
```

Limitations, cont'd

◆ Analysis

1. Let N be the actual length of `string`.
2. The loop goes from $0 \dots (N-1)$, so N -times.
3. During each time, `strlen(string)` is called.
4. `strlen(string)` also has to go over the N characters of `string`.
5. Each `strlen(string)` costs at least N operations, there are N calls to `strlen(string)`, therefore there are at least $N*N$ operations (we call it $O(N^2)$)

Limitations, cont'd

- ◆ **Enough silliness!** *Optimize that sucker!*
 - ◆ Simple enough . . .

```
void smartUppercase (char* string)
{
    int i;
    int length = strlen(string);
    for (i = 0; i < length; i++)
        string[i] = toupper(string[i]);
}
```

Not so fast, *bucko!*

- ◆ You (as the compiler) have committed a mortal sin . . .
- ◆ ***You have changed the algorithm!***
 - ◆ Old: $O(N^2)$, New: $O(N)$
 - ◆ Why is this so bad?
- ◆ So, if the compiler can not be relied upon to do this type of optimization ***whose responsibility is it?***

Limitations, example

- Consider an abstract data-type called `vector` with the following interface:

```
class Vector
{
    int length;
    int* dataPtr;
public:
    Vector (int len)
        { length=len; dataPtr = new int[len]; }
    int  getLength () { return(length); }
    int* getDataPtr() { return(dataPtr); }
    int  getElement (int i, int* dest)
        { if (i<0||i>=length) return(0);
          *dest = dataPtr[i];
          return(1);      }
};
```

Limitations, example:

- ◆ Say we want to sum all elements in vector
 - ◆ Original code:

```
int sum (Vector* vPtr)
{
    int eleVal;
    int s = 0;
    for (int i = 0; i < vPtr->getLength(); i++)
    {
        vPtr->getElement(i, &eleVal);
        s += eleVal;
    }
    return s;
}
```

Limitations, example, cont'd:

- Well, don't call `getLength()` all those times:
 - Code motion:

```
int sum (Vector* vPtr)
{
    int eleVal;
    int s = 0;
    int length = vPtr->getLength();
    for (int i = 0; i < length; i++)
    {
        vPtr->getElement(i, &eleVal);
        s += eleVal;
    }
    return s;
}
```

Limitations, example, cont'd:

- ◆ Don't call `getElement ()` all those times:
 - ◆ Reduction in strength:

```
int sum (Vector* vPtr)
{
    int eleVal;
    int s = 0;
    int length = vPtr->getLength();
    int* iPtr = vPtr->getDataPtr();
    for (int i = 0; i < length; i++)
    {
        eleVal = *(iPtr+i);
        s += eleVal;
    }
    return s;
}
```


Limitations, example, cont'd:

- ◆ Eliminate obvious inefficiency:

```
int sum (Vector* vPtr)
{
    int s = 0;
    int length = vPtr->getLength();
    int* iPtr = vPtr->getDataPtr();
    for (int i = 0; i < length; i++)
        s += *(iPtr+i);
    return s;
}
```

There you have it: optimized!

- ♦ But at what cost!
 - ♦ Look at this! *Is this good software engineering?*
 - ♦ Is it good object-oriented design?
 - ♦ What is fundamentally a *better solution?*

```
int sum (Vector* vPtr)
{
    int s = 0;
    int length = vPtr->getLength( );
    int* iPtr = vPtr->getDataPtr( );
    for (int i = 0; i < length; i++)
        s += *(iPtr+i);
    return s;
}
```

Another problem: aliasing!

- Two different expressions refer to same memory:

- Example 1:

```
int    i = 10;  
int*   iPtr = &i;  
(*iPtr)++; i--;
```

- Example 2:

```
vPtr->getElement(2, &eleVal1);  
eleVal2 = *(vPtr->getDataPtr() + 2);
```

You control the optimizing:

For gnu compilers for Linux (`gcc` and `g++`):

- `-O0` (No optimization)
 - Best for debugging because there's a better correlation between assembly language code and source code
 - This is the default
- `-O1` (Optimize)
 - Takes more compiler time
- `-O2` (Optimize even more)
 - Takes even more compiler time but does almost all that do not involve a space-speed tradeoff
- `-O3` (Optimize even *more!*)
 - Takes even more
- `-Os` (Optimize for size)

Profiling

- ◆ Just how fast is it?
 - ◆ Link with `-pg` flag: inserts code that times fncs
 - ◆ Running program creates file `gmon.out` w/timing info
 - ◆ Unix tool `gprof <program>` uses `gmon.out` to display timing info for fncs in `<program>`.
 - ◆ Example: `testGprof` is a bubblesort of 10,000 ints:
 - ◆ Two nested loops (therefore $O(N^2)$)
 - ◆ Inner loop exchanges pairs ($O(1)$ time)
 - ◆ Outer loop does inner loop while at least one pair exchanged

Profiling, cont'd

```
[jphillips@localhost lecture1]$ gprof testGprof
Flat profile:
```

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ns/call	total ns/call	name
78.80	0.51	0.51				__libc_csu_init
21.63	0.65	0.14	24883341	5.65	5.65	main

%
time the percentage of the total running time of the
program used by this function.

cumulative
seconds a running sum of the number of seconds accounted
for by this function and those listed above it.

self
seconds the number of seconds accounted for by this
function alone. This is the major sort for this
listing.

calls the number of times this function was invoked, if
this function is profiled, else blank.

“So, how do I actually program knowing all this?”

1. It's up to ***you*** to optimize across ***function calls***
2. Compilers have limited ability to detect ***common expressions, do that yourself too***
3. Compilers are reasonably good at reduction in strength, ***therefore write for clarity to other programmers!***
4. ***Use local vars and accumulate within loops*** to tell compiler not to worry about aliasing
5. When in doubt look at the assembly (***“Eeww!”***) or ***profile*** to see ***who*** the big time hogs are

Next time: *Linking!*

