CSC 374/407: Computer Systems II

Lecture 1

Joseph Phillips
De Paul University

2017 September 5

Reading

- Bryant & O'Hallaron "*Computer Systems, 3rd 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 review some C:

A little C (Output)

- Output in C with **`printf()`** ("print-formatted")
  - **`printf("template",expr`**$_1$**`,...expr`**$_n$**`)`**
- 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)

But printf()'s man page says:

int printf(const char *format, ...);

So that means I should write:

```
#include <stdlib.h>
#include <stdio.h>
int main ()
{
  int printf(const char* "Hello world\n");
  return(EXIT_SUCCESS);
}
```

Right?

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)
```

(***STR***ing ***TO*** ***L***ong)

```
strtod
(const char* text,
 char**     endPtr)
```

(***STR***ing ***TO*** ***D***ouble)

- **`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: ***<u>addresses and pointers! (1)</u>***

- What is an address?
  - It is where in memory some object starts.
  - All objects in memory have unique starting addresses
  - In general <u>you</u> do ***not*** choose numbers, the <u>OS/compiler/running system</u> 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: ***<u>addresses and pointers! (3)</u>***

- 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,
     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`?

```
int i,j,n;
```

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:
  - Where they point can be changed by ++, --
- 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++;
(*intPtr) *= 10;

// What will this print?
printf("%d\n",intArray[1]);
```

Const pointers

- ***`const`*** **`SomeType* ptr;`**
- Same promise as **`const`** references:
  - *"I, the programmer, promise not to change the pointed to object"*
  - You may ***look at*** the object, but you may ***not touch!***
  - 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!

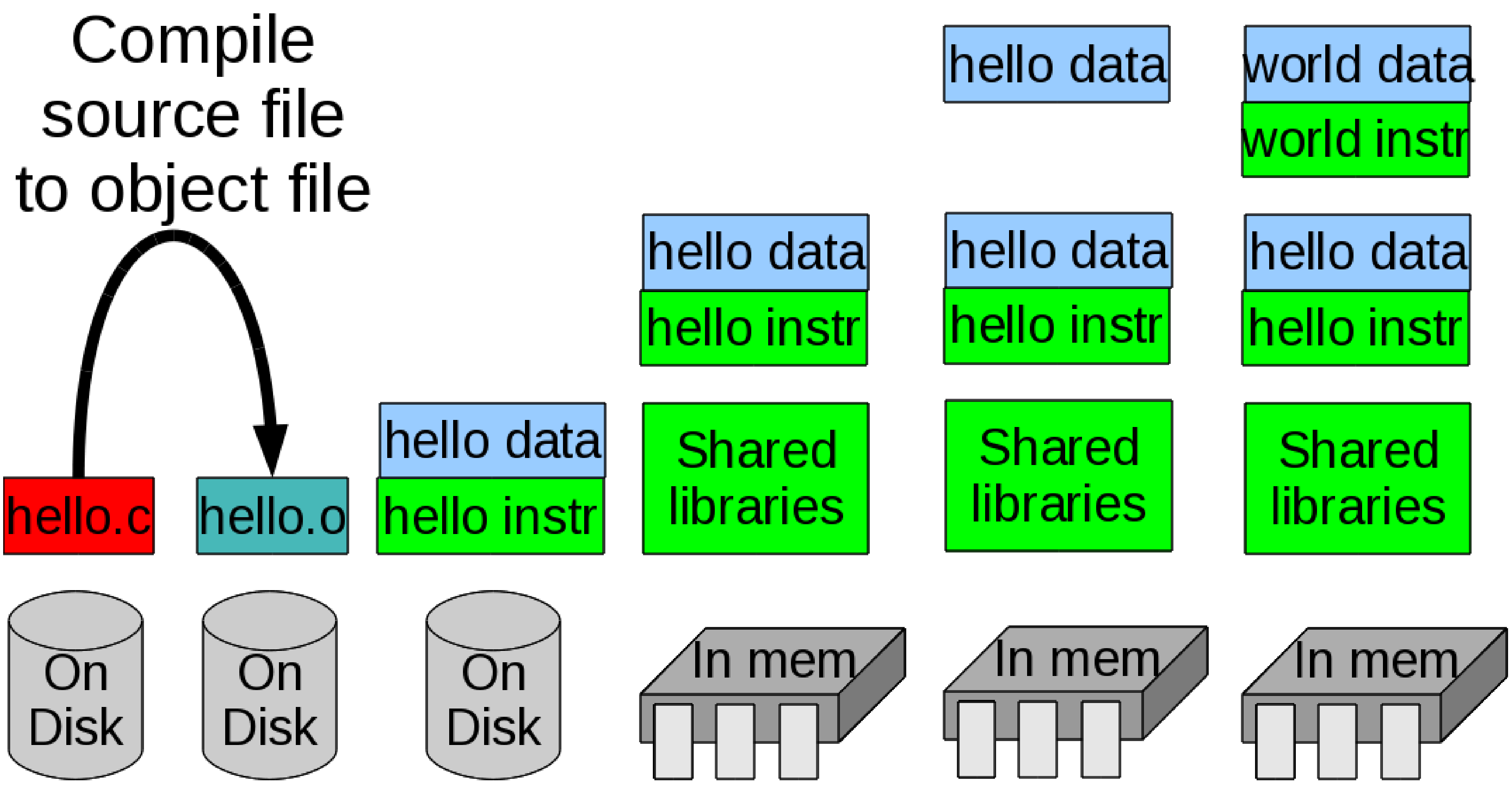
- Happy compiler or sad compiler ?

```
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?
}
```

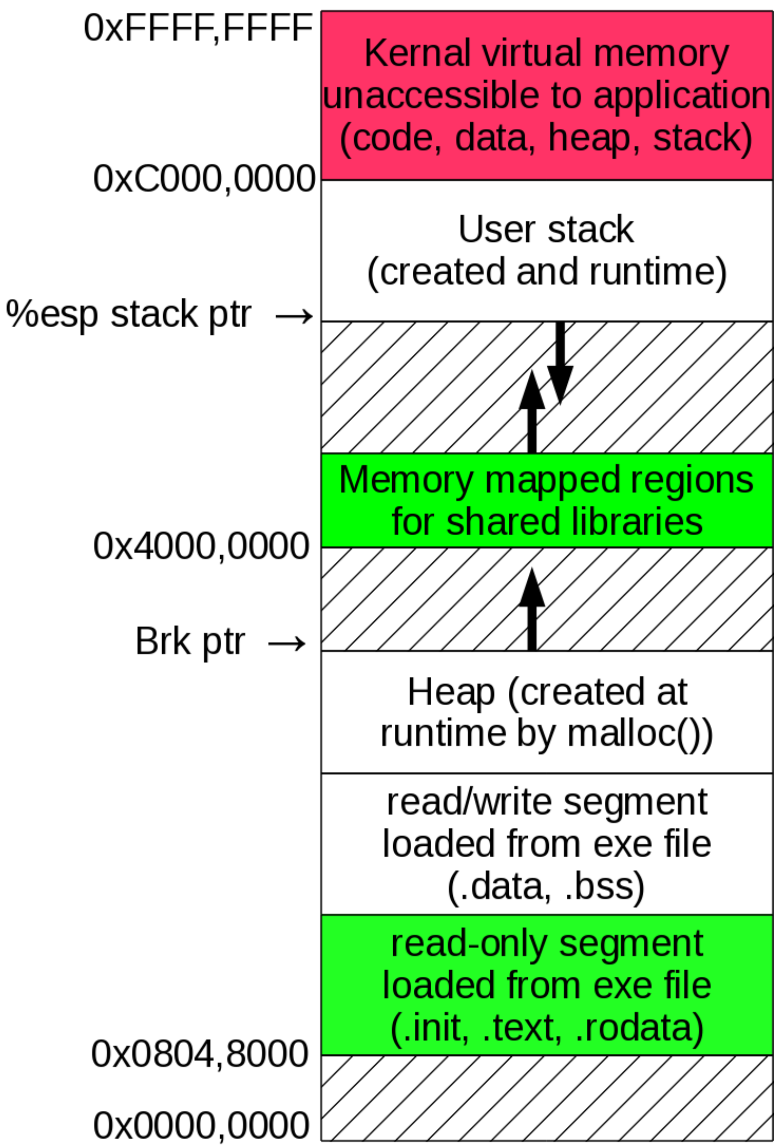
Today's topic (in time)

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 <u>this</u> lecture!***

Speed it up! (3)

- We will study machine independent optimizations
  - Make sense to do no matter what your CPU
- 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! (4)

- 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++ -O*__0__*

```
  0:  push %rbp
  1:  mov  %rsp,%rbp
  4:  mov  %edi,-0x14(%rbp) edi=limit
  7:  movl $0x0,-0x4(%rbp)  sum = 0
  e:  movl $0x0,-0x8(%rbp)  i   = 0
 15:  jmp  21 <counter+0x21>
 17:  mov  -0x8(%rbp),%eax  sum += i
 1a:  add  %eax,-0x4(%rbp)  (ditto)
 1d:  addl $0x1,-0x8(%rbp)  i++
 21:  mov  -0x8(%rbp),%eax
 24:  cmp  -0x14(%rbp),%eax
 27:  jle  17 <counter+0x17> i<=limit
 29:  mov  -0x4(%rbp),%eax
 2c:  pop  %rbp
 2d:  retq
```

Using Registers (2)

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

<u>With</u> optimization some variables are kept in registers

g++ -O***<u>2</u>***

```
 0: test   %edi,%edi
 2: js     1b <counter+0x1b> limit == 0?
 4: add    $0x1,%edi
 7: xor    %edx,%edx         i   = 0
 9: xor    %eax,%eax         sum = 0
 b: nopl   0x0(%rax,%rax,1)
10: add    %edx,%eax         sum += i
12: add    $0x1,%edx         i++
15: cmp    %edi,%edx
17: jne    10 <counter+0x10>
19: repz retq
1b: xor    %eax,%eax
1d: retq
```

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]
}
```

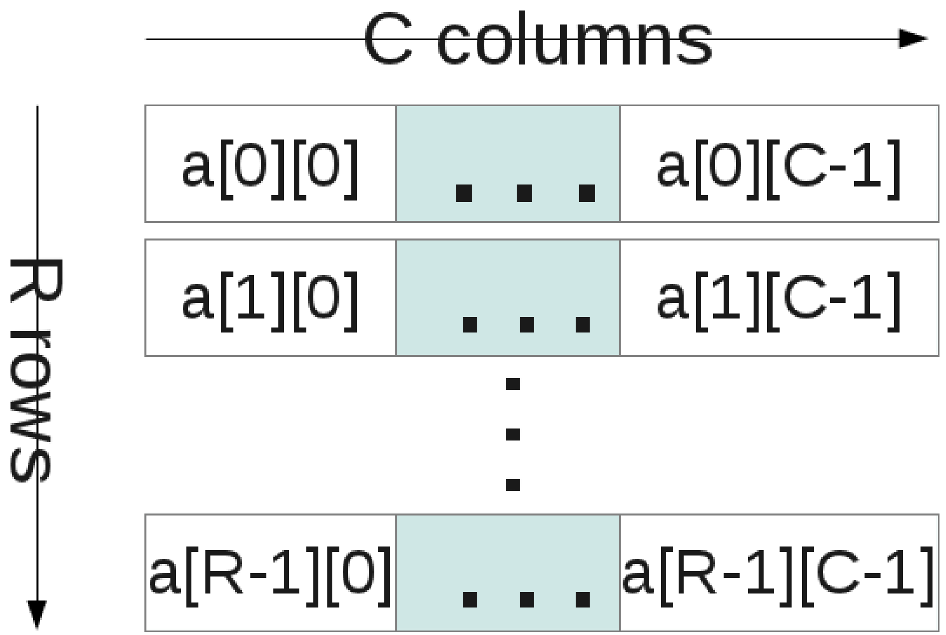
Your turn!

- 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];
```

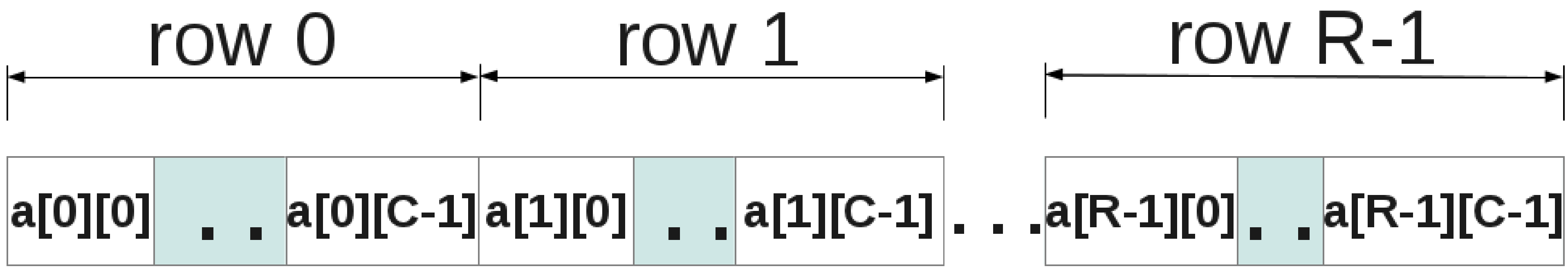
Compute common expressions once

- 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
- `&array2D[i][j] == &array1D[i*NumCols + j]`

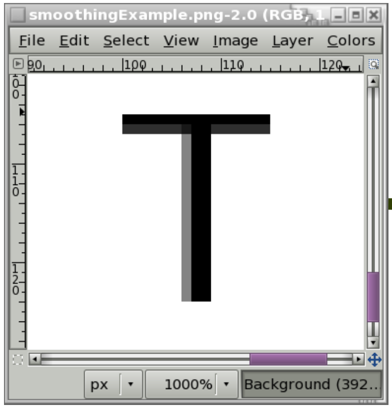


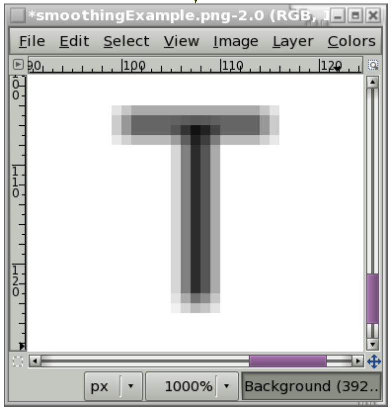
Compute common expressions once

- 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!

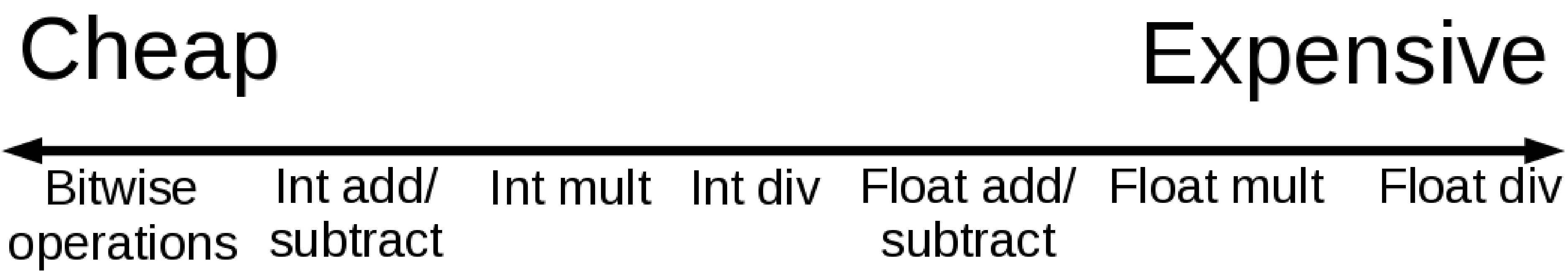
- Optimize:

```
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
- "`i*10`" can 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

- It makes sense to do:
  - bitwise and (&) instead of modulus (%) by powers of 2
- Consider (`i % 8`)
  - is as costly as an integer division
  - The result will be in {0,1,2,3,4,5,6,7} (the last 3 bits)
- Now consider (`i & 0x7`)
  - Isolates last 3 bits more cheaply

Reduction in strength, ex

```
unsigned int num=strtol("12",0,0);
//   callq  21 <unsignedInt+0x21>
//   mov    %eax,-0x4(%rbp)

unsigned int numDiv4 = num / 4;
//   mov    -0x4(%rbp),%eax
//   shr    $0x2,%eax
//   mov    %eax,-0x8(%rbp)

unsigned int numMul4 = num * 4;
//   mov    -0x4(%rbp),%eax
//   shl    $0x2,%eax
//   mov    %eax,-0xc(%rbp)
```

Reduction in strength, ex 2

Trickier dance if dealing with signed numbers:

```
int num = strtol("1234",0,0);
//    callq  59 <signedInt+0x21>
//    mov    %eax,-0x4(%rbp)

int numDiv4 = num / 4;
//    mov    -0x4(%rbp),%eax
//    lea    0x3(%rax),%edx
//    test   %eax,%eax
//    cmovs  %edx,%eax
//    sar    $0x2,%eax
//    mov    %eax,-0x8(%rbp)
```

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 expen$ive 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 O0 (none)

```
      . . .
3f:  movl    $0x0,-0x4(%rbp)    i = 0
46:  jmp     98 <main+0x98>
48:  movl    $0x0,-0x8(%rbp)    j = 0
4f:  jmp     8c <main+0x8c>
51:  mov     -0xc(%rbp),%eax    eax = n
54:  imul    -0x4(%rbp),%eax    eax = n*i
58:  mov     %eax,%edx          edx = n*i
5a:  mov     -0x8(%rbp),%eax    eax = j
5d:  add     %edx,%eax          eax = n*i + j
5f:  cltq
61:  lea     0x0(,%rax,4),%rdx  rdx = 4*(n*i+j)
69:  mov     -0x18(%rbp),%rax   rax = &a[0]
6d:  add     %rax,%rdx          rdx = &a[n*i+j]
70:  mov     -0x8(%rbp),%eax    eax = j
73:  cltq
75:  lea     0x0(,%rax,4),%rcx  rcx = j*4
7d:  mov     -0x20(%rbp),%rax   rax = &b[0]
81:  add     %rcx,%rax          rax = &b[j]
84:  mov     (%rax),%eax        eax = b[j]
86:  mov     %eax,(%rdx)        a[n*i+j] = b[j]
88:  addl    $0x1,-0x8(%rbp)    j++
      . . .
```

Optimize O1 (First level)

```
 0:  mov    $0x64,%edx
 5:  jmp    11 <main+0x11>
 7:  sub    $0x1,%eax
 a:  jne    7 <main+0x7>
 c:  sub    $0x1,%edx
 f:  je     18 <main+0x18>
11:  mov    $0x64,%eax
16:  jmp    7 <main+0x7>
18:  mov    $0x0,%eax
1d:  retq
```

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`** . . (**`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($\mathbf{N}^2$), New: O($\mathbf{N}$)
  - 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>`**.
    - ***Remember*** to scroll up to the beginning of the output.
  - Example: **`testGprof`** is a bubblesort of 10,000 ints:
    - Two nested loops (therefore O($\mathbf{N}^2$))
    - Inner loop exchanges pairs (O(**`1`**) time)
    - Outer loop does inner loop while at least one pair exchanged

Profiling, cont'd

```
[jphill13@cdmlinux Lecture01]$ gcc bubbleSort.c -o bubbleSort -pg
[jphill13@cdmlinux Lecture01]$ ./bubbleSort
[jphill13@cdmlinux Lecture01]$ gprof bubbleSort
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 78.57     14.50    14.50        1    14.50    18.35  bubbleSort
 20.87     18.35     3.85 623881099     0.00     0.00  exchange
  1.48     18.62     0.27                               frame_dummy

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

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

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

“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

I don’t have time, but also check out:

- gdb: The GNU debugger

Next time: *Linking!*