Review

- Dynamic memory allocation
  - Look a-like dynamic 2D array
  - Simulated 2D array
  - How cache memory / cache line works
- Command line arguments
- Pre-processor directives
  - #define
  - #ifdef … #else … #endif
  - #ifndef … #else … #endif

Lecture 3

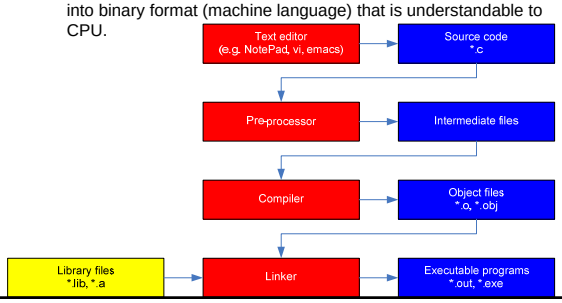
Advanced Programming Topics

Topics

- Defining macros through compilation flags
- External Libraries & Functions
- Makefile
- Source code optimization

Review: How executable files are generated

- Compiled
  - Programs written in C language are translated through compiler into binary format (machine language) that is understandable to CPU.



Defining macros through compilation flags

1. Define DATA type outside the source code

/home/course/Lecture03/01.cpp

```
#include <iostream>
using namespace std;
int main() {
  DATA a[100];
  cout << "a occupies " << sizeof(a)
            << " bytes." << endl;
  return 0;
}
```

g++ 01.cpp -D DATA=int
g++ 01.cpp -D DATA=double
g++ 01.cpp -D DATA="unsigned char"

2. Use with conditional compilation

/home/course/Lecture03/02.cpp

```
#include <iostream>
using namespace std;

int main() {
  #ifdef LIMIT
    cout << "The trial version supports 10 integers.";
    int data[10];
  #else
    cout << “The full version supports 200000 integers.";
    int data[200000];
  #endif
  data[0] = 5;
  return 0;
}
```

g++ 02.cpp
g++ 02.cpp –DLIMIT

7

External Functions & Libraries

8

/home/course/Lecture03/03.cpp

```
#include "stopWatch.h"
int count=0;
double t=0;
double y, ang=24.0;
stopWatch timer;
do {
  timer.start();
  y=log10(ang);
  timer.stop();
  t += timer.elapsedTime();
  count++;
} while(t<0.1);

std::cout << "\nThis machine can do " << ((double)
 count) / t << " log10 operations per second. ";
```

9

External Functions & Libraries

- Having a 03.cpp, stopWatch.h, and stopWatch.o
- **Method 1: (N/A)**
  - g++ –Wall 03.cpp stopWatch.cpp –o 03.exe
  - SOURCE CODE NOT PROVIDED!
- **Method 2:**
  - g++ -c stopWatch.cpp
  - g++ -Wall 03.cpp stopWatch.o –o 03.exe
  - Note that the order IS important

10

External Functions & Libraries

- **Method 3:**
  - g++ –c stopWatch.cpp
  - ar –q libMylib.a stopWatch.o
    Create a statically linked library “libMyLib.a” with stopWatch.o in the library
  - g++ –Wall 03.cpp libMylib.a –o 03.exe
    g++ –Wall 03.cpp -lMylib –o 03.exe
    g++ –Wall 03.cpp –lMylib –LmyPath –o 03.exe

11

Makefile

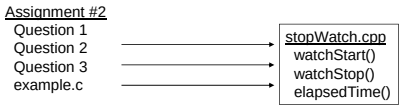
12

Makefile

- Makefile is a kind of script to describe how to compile a program or a series of related programs
- Nowadays make is mostly GNU Make
  - http://www.gnu.org/software/make/make.html
  - Make is a tool which controls the generation of executables and other non-source files of a program from the program's source files.
- Command 'make' in UNIX/Linux first searches the current directory for makefile or Makefile to generate your code
  - You may specify alternative makefiles by using 'make -f *mymakefile*'

13

Scenario




```
g++ q2.cpp stopWatch.c –o q2.exe
g++ q3.cpp stopWatch.c –o q3.exe
g++ example.cpp stopWatch.c –o example.exe
```

```
g++ stopWatch.c
g++ q2.cpp stopWatch.o –o q2.exe
g++ q3.cpp stopWatch.o –o q3.exe
g++ example.cpp stopWatch.o –o example.exe
```

14

Example

```
g++ stopWatch.c
g++ q2.cpp stopWatch.o –o q2.exe
g++ q3.cpp stopWatch.o –o q3.exe
g++ example.cpp stopWatch.o –o example.exe
```

```
Target: source file(s)
        command (must be preceded by a tab)
```

```
all: q2.exe q3.exe example.exe

q2.exe: q2.cpp stopWatch.o
[TAB]   g++ q2.cpp stopWatch.o –o q2.exe

q3.exe: q3.cpp stopWatch.o
[TAB]   g++ q3.cpp stopWatch.o –o q3.exe

example.exe: example.cpp stopWatch.c
[TAB]   g++ example.cpp stopWatch.o –o example.exe

stopWatch.o: stopWatch.c
[TAB]   g++ –c stopWatch.c
```

15

Example

```
# Define compiler & flags
CC=g++
CFLAGS=-Wall -O2

# Define projects to be built
all: q2.exe q3.exe example.exe

# Defines rules for compiling the project
q2.exe: q2.cpp stopWatch.o
	$(CC) $(CFLAGS) q2.cpp stopWatch.o –o q2.exe

q3.exe: q3.cpp stopWatch.o
	$(CC) $(CFLAGS) q3.cpp stopWatch.o –o q3.exe

example.exe: example.cpp stopWatch.o
	$(CC) $(CFLAGS) example.cpp stopWatch.o –o example.exe

stopWatch.o: stopWatch.c
	$(CC) $(CFLAGS) –c stopWatch.c
```

16

Example - Macros

```
# Define compiler & flags
CC=g++
CFLAGS=-Wall -O2

OBJS=stopWatch.o
all: q2.exe q3.exe example.exe

# Defines rules for compiling the project
q2.exe: q2.cpp $(OBJS)
	$(CC) $(CFLAGS) q2.cpp $(OBJS) –o q2.exe

q3.exe: q3.cpp $(OBJS)
	$(CC) $(CFLAGS) q3.cpp $(OBJS) –o q3.exe

example.exe: example.c $(OBJS)
	$(CC) $(CFLAGS) example.cpp $(OBJS) –o example.exe

stopWatch.o: stopWatch.c
	$(CC) $(CFLAGS) –c stopWatch.c
```

17

Special macros

CC: Contains the current C compiler. Defaults to cc.

CFLAGS: Special options which are added to the built-in C rule.

$@: Full name of the current target.

$?: A list of files for current dependency which are out-of-date.

$<: The source file of the current (single) dependency.

18

Example – Special macros

```
# Define compiler & flags
CC=g++
CFLAGS=-Wall –O2

OBJS=stopWatch.o
all: q2.exe q3.exe example.exe

# Defines rules for compiling the project
q2.exe: q2.cpp $(OBJS)
	$(CC) $(CFLAGS) $< $(OBJS) –o $@

q3.exe: q3.cpp $(OBJS)
	$(CC) $(CFLAGS) $< $(OBJS) –o $@

example.exe: example.cpp $(OBJS)
	$(CC) $(CFLAGS) $< $(OBJS) –o $@

stopWatch.o: stopWatch.c
	$(CC) $(CFLAGS) –c $<
```

19

Example – Implicit Rules

```
# Define compiler & flags
CC=g++
CFLAGS=-Wall –O2

OBJS=stopWatch.o
all: q2.exe q3.exe example.exe

%.exe: %.cpp $(OBJS)
	$(CC) $(CFLAGS) $< $(OBJS) –o $@
%.o: %.c
	$(CC) $(CFLAGS) –c $<
```

20

Source Code Optimization

21

Data Type Consideration

- Integer
  - – unsigned:
    - Division and remainders
    - Loop counters
    - Array indexing
  - – signed:
    - Integer to float conversion
- In function prototypes
  - – Use const as much as possible
    - By the way, what is a constant pointer ?

22

Constant pointer

```
#include <iostream>
using namespace std;

int main() {
   const int *a;
   int p[] = {1, 2, 3};
   int q[] = {4, 5, 6};
   int i;

   a = p;
   for(i=0;i<3;i++) {
        cout << a[i];
        a[i] = 0;
   }
```

```
   a = q;
   for(i=0;i<3;i++) {
        cout << a[i];
        a[i] = 0;
   }
   return 0;
}
```

23

Which line above will cause a compilation error?

Array Access and Loop Optimization

- Loop jamming / Loop fusion
- Loop fission / Loop distribution
- Move invariants out of loops
  - – Loop un-switching
- Loop peeling
- Loop interchange
- Loop unrolling
- Loop unrolling and sum reduction

24

Loop Jamming / Loop Fusion

```
for(i=0;i<N;i++) {
  b[i] += 1;
}
for(i=0;i<N;i++) {
  y += b[i];
}
```

```
for(i=0;i<N;i++) {
  b[i] += 1;
  y += b[i];
}
```

Pentiun D930, icpc -O2, N=10000000
Separated: 0.133132 secs.
Fused: 0.116919 secs.

25

Loop Fission / Loop Distribution

```
for(i=0;i<N;i++) {
  x += b[i];
  y += c[i];
}
```

```
for(i=0;i<N;i++)
  x += b[i];
for(i=0;i<N;i++)
  y += c[i];
```

Pentiun D930, icpc -O2, N=1000000
Fused: 1.3226511 secs.
Separated: 0.00791911 secs.

26

Move invariants out of loops
If() - Loop unswitching

```
for(i=0;i<N;i++) {
  for(j=0;j<N;j++) {
    if(a[i] > 100.0) b[i] = a[i] - 3.7;
    x = x + a[j] + b[i];
  }
}
```

```
for(i=0;i<N;i++) {
  if(a[i] > 100.0) b[i] = a[i] - 3.7;
  for(j=0;j<N;j++) {
    x = x + a[j] + b[i];
  }
}
```

Pentiun D930, icpc -O2, N=10000
Inside: 0.256424 secs.
Outside: 0.203336 secs.

27

Move invariants out of loops
If() - Loop unswitching

```
for(i=0;i<N;i++) {
   x[i] = x[i] + y[i];
   if(w) z[i]=0.0;
}
```

```
if(w) {
   for(i=0;i<N;i++) {
     x[i] = x[i] + y[i];
     z[i]=0.0;
   }
}
else {
   for(i=0;i<N;i++) {
     x[i] = x[i] + y[i];
   }
}
```

Pentiun D930, icpc -O2, N=1000000
Before: 0.022519 secs.
After : 0.021649 secs.

28

Move invariants out of loops
Math operations

```
for(i=0;i<N;i++) {
  a[i] = 0.0;
  for(j=0;j<N;j++) {
    a[i] += b[j] * d[j] * c[i];
  }
}
```

```
for(i=0;i<N;i++) {
  a[i] = 0.0;
  for(j=0;j<N;j++) {
    a[i] += b[j] * d[j];
  }
  a[i] *= c[i];
}
```

Pentiun D930, icpc -O2, N=10000
Inside: 0.274137 secs.
Outside: 0.200256 secs.

29

Loop Peeling

```
j = N-1;
for(i=0;i<N;i++) {
  b[i] = (a[i] + a[j]) * 0.5;
  j = i;
}
```

```
b[0] = (a[0] + a[N-1]) * 0.5;
for(i=1;i<N;i++) {
  b[i] = (a[i] + a[i-1]) * 0.5;
}
```

Pentiun D930, icpc -O2, N=1000000
Before: 0.011461 secs.
After : 0.011407 secs.

30

Loop Interchange
Stride minimization

```
for(i=0;i<N;i++) {
  for(j=0;j<N;j++) {
    c[i][j] += a[i][j] + b[i][j];
  }
}
```

```
for(j=0;j<N;j++) {
  for(i=0;i<N;i++) {
    c[i][j] += a[i][j] + b[i][j];
  }
}
```

Pentiun D930, icpc -O2, N=300
Stride 1: 0.001437 secs.
Stride N: 0.002837 secs.

Loop Interchange
Exercise

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


Loop Unrolling

```
for(i=0;i<N;i++)
  for(j=0;j<N;j++)
    for(k=0;k<4;k++)
      a[i][j] += b[k][i] * c[k][j];
```

```
for(i=0;i<N;i++) {
  for(j=0;j<N;j++) {
    a[i][j] += b[0][i] * c[0][j];
    a[i][j] += b[1][i] * c[1][j];
    a[i][j] += b[2][i] * c[2][j];
    a[i][j] += b[3][i] * c[3][j];
  }
}
```

Pentiun D930, icpc -O2 -unroll0, N=2000
Before: 2.07285 secs.
After : 0.069612 secs.

Loop Unrolling and Sum Reduction

```
for(i=0;i<N;i++)
  for(j=0;j<N;j++)
    for(k=0;k<4;k++)
      a += b[k][i] * c[k][j];
```

```
for(i=0;i<N;i++) {
  for(j=0;j<N;j++) {
    a += b[0][i] * c[0][j];
    a += b[1][i] * c[1][j];
    a += b[2][i] * c[2][j];
    a += b[3][i] * c[3][j];
  }
}
```

Pentiun D930, icpc -O2 -unroll0, N=2000
Before: 1.99077 secs.
After : 0.039288 secs.

Loop Unrolling and Sum Reduction

```
for(i=0;i<N;i++) {
  for(j=0;j<N;j++) {
    a1 += b[0][i] * c[0][j];
    a2 += b[1][i] * c[1][j];
    a3 += b[2][i] * c[2][j];
    a4 += b[3][i] * c[3][j];
  }
}
a = a1 + a2 + a3 + a4;
```

Pentiun D930, icpc -O2 –unroll0, N=2000
Before: 1.99077 secs.
After : 0.039288 secs.
Further: 0.029734 secs.

Write Efficient Code

- Contiguous memory access helps to improve code efficiency. (why?)
- Loop unrolling & SIMD instructions (e.g. MMX, SSE, 3DNOW, …) can be used to help vector operations.
- Modern compilers (gcc, Intel compiler) can do loop unrolling automatically for you.
  - icc *-O2 -unroll -axP* yourcode.cpp
  - gcc *-O2 -funroll-loops -march=p4 -msse3* yourcode.cpp
- Compiler flags (red text above) can make a huge different in terms of both computational efficiency and sometimes results.
- Use assembly language to gain ultimate control on the instruction flow to processors.

Matrix operations

- For operations involve matrices, each element may be referenced more than once, thus provides better chance for optimizing memory access pattern (reuse the data that is already in cache memory)
- Even if elements (either vector or matrix) are referenced only once, how the memory is accessed still dictates the performance.
  - We must pay attention to the 2-D data layout in memory.
- As modern computers has blocking memory access (lecture one, cache-line: one memory request returns a block of memory), we need to work on adjacent entries first.

37