CSE 403
Lecture 16

Coding

Coding for comprehensibility

```
#include <stdio.h>
char *T="IeJKLMaYQ[illegible]<:90!\"$434-./2>]s",
K[3][1000],*F,x,A,*[illegible],r[4],*g,N,Y,*Q,W[illegible]X(){r [r [r[3]=M[1-
(x&1)][*r=W,1],2[illegible]1]=x+1+Y,*g++=((([illegible]-1)>>1)-
1)?*r:r[x>>3],([illegible]();}E(){A||X(x=0,g [illegible]7&(*T>>A*3),J[(x[F]-
W-x)^A*7]=Q[[illegible]A[illegible]+( x&1)],g=J+((x[[illegible]A*7)-
A,g[1]=(*M)[*[illegible]T+=A[illegible]1],x&1],(A^=1)&&[illegible]+=W);}I(){E(--q&&I
() );}B(){*J&&[illegible]=*J,Q[2[illegible]<k[1]&&(*g++=[illegible]D-W&&D-9&&D-
10&&D-13)&&[illegible]&(*g++=[illegible]1)||64<D&&D<[illegible]*r=0,*g++=D-
63)||D >= 97[illegible]<123&&(*r=[illegible]=D-95)||!(D-k[illegible]
)&&(*r=0,*g[illegible]2)||D>k[3]&&D[illegible] -1&&(*r=[illegible]+=D-47),J++));}j(
){ putchar(A),[illegible](j(A=(*K)[D* W+[illegible]+x]),++x[illegible]&b();}t ()
{(j((b(D=q[g],[illegible]A=W) ), ++q<(*([illegible]2*(r+1[illegible]&&t();}R(){(A=(t( q=
0),'\n'),j(),++r[illegible])&&R();}O() {( j(([illegible]R( )[illegible]-=q) && O(g-=-q) ;}
C(){( J= gets (K[illegible]&&C((B(g=K[2]),*r=[illegible]=0)),(*r)[r]=g-
K[2],g=K[2 ],r[ 1[illegible])) );;} main (){C ((I([illegible][K], A[M] =(F= (k=(
M[!A ]=(Q =T+( q[illegible]= 32)- (N=4 )))) [illegible]+7 )+7) ),Y= N<<( *r=! -
A)) );;}
```

Can code be self documenting?

- Incorrect comments are worse than missing comments
- Comments should not repeat what is clear from the code
- Code should be written to minimize the need for comments
- Code that is too complicated to explain should be rewritten

Oops

```
// This whole thing is too complicated for me to understand or explain, but
// here is where the actual work takes place,  I think.
private void ListenerWorker(RTPListener.RTPStream rtpStream){
        try {
                rtpListener.Subscribe(rtpStream.SSRC);
                Listen(rtpStream);
        }
        catch (System.Exception se){
                LogEvent(se.ToString(), EventLogEntryType.Error);
        }
}
```

Commenting

- Comment data declarations, including units and ranges
- Comment meanings of control structures
- Avoid commenting structures that are difficult to maintain
- Write comments before, or while coding, not afterwords!

memcpy

```
/* memcpy -- copy a block of size bytes from pvFrom
      to pvTo */

void *memcpy(void *pvTo, *void pvFrom, size_t size){









   return pvTo;

}
```

memcpy 0

```
void *memcpy(void *pvTo, void *pvFrom, size_t size)
{
   byte *pbTo =   (byte *)pvTo;
   byte *pbFrom = (byte *)pvFrom;

   while (size-- > 0)
      *pbTo++ = *pbFrom++;
   return (pvTo);
}
```

memcpy I

```
void *memcpy(void *pvTo, void *pvFrom, size_t size)
{
   byte *pbTo =   (byte *)pvTo;
   byte *pbFrom = (byte *)pvFrom;
   if (pvTo == NULL || pvFrom == NULL)
   {
      fprintf(stderr, “Bad args in memcpy\n”);
      abort();
   }
   while (size-- > 0)
      *pbTo++ = *pbFrom++;
   return (pvTo);
}
```

No errors here...

- ..but it’s bigger and slower

- So, exploit the preprocessor

memcpy II

```
void *memcpy(void *pvTo, void *pvFrom, size_t size)
{
   byte *pbTo =   (byte *)pvTo;
   byte *pbFrom = (byte *)pvFrom;
   #ifdef DEBUG
   if (pvTo == NULL || pvFrom == NULL)
   {
      fprintf(stderr, “Bad args in memcpy\n”);
      abort();
   }
   #endif
   while (size-- > 0)
      *pbTo++ = *pbFrom++;
   return pvTo;
}
```

memcpy III

```
void *memcpy(void *pvTo, void *pvFrom, size_t size)
{
   byte *pbTo =   (byte *)pvTo;
   byte *pbFrom = (byte *)pvFrom;

   assert(pvTo != NULL && pvFrom != NULL);

   while (size-- > 0)
      *pbTo++ = *pbFrom++;
   return pvTo;
}
```

- Assertions can be turned on and off
  - You probably shouldn’t consider rewriting the assert macro

memcpy IV

```
void *memcpy(void *pvTo, void *pvFrom, size_t size)
{
   byte *pbTo =   (byte *)pvTo;
   byte *pbFrom = (byte *)pvFrom;

   assert(pvTo != NULL && pvFrom != NULL);
   assert(pbTo >= pbFrom+size || pbFrom >= pbTo+size);

   while (size-- > 0)
      *pbTo++ = *pbFrom++;
   return pvTo;
}
```

assertions

- Don’t use assertions to check unusual conditions
  - You need explicit error code for this
- Only use them to ensure that illegal conditions are avoided

Memory

- The memcpy examples are from *Writing Solid Code: Microsoft's Techniques for Developing Bug-Free C Programs*
- Although the book is general, lots of the guidelines focus on memory issues
  - Marking freed memory
  - Not accessing freed memory
  - Dealing with details of `realloc`
- These are real issues, but appear less frequently in other languages

Writing solid code

- Shred your garbage

```
void FreeMemory(void *pv){
        Assert(pv != NULL);
        memset(pv, 0xA3, sizeofBlock(pv);
        free(pv);
}
```

- Force early failure, increase determinism
- Why 0xA3?

Should debug code be left in shipped version

- Pro:
  - Debug code useful for maintenance
  - Removing debug code change behavior
    - Bugs in release but not debug versions
- Con:
  - Efficiency issues
  - Different behavior for debug vs. release
    - Early fail vs. recover