High Performance Computing Lecture 2

Alexandros Stamatakis
Junior Research Group Leader

The Exelixis Lab
Department of Computer Science
Technical University of Munich

stamatak@cs.tum.edu
http://icwww.epfl.ch/~stamatak

Introduction to C

```
#include <stdio.h>
/* All time classic hello world */
int main(int argc, char **argv)
{
  printf("Hello World\n");
  return 0;
}
```

gcc –Wall –g my_program.c –o my_program

Can't stress this often enough, especially for beginners in C: TURN THE WARNINGS ON!

Generates DEBUG INFO: remove when compiling for performance!

Intro to C

Can a program have more than one .c file?

#include inserts a ".h" files "header" files. They contain e.g. function declarations/interfaces to libraries and code in other ".c" files.

What do the < > mean?

This is a comment.

The main() function is always where your program starts running.

```
#include <stdio.h>
/* This is a comment */
int main(int argc, char **argv)
{
  printf("Hello World\n");
  return 0;
}
```

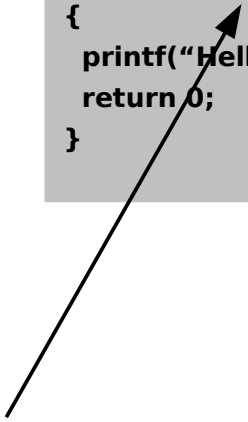
lexical scopes are delimited by { … }

Print out a message. '\n' means "new line".

main function should always return '0' upon error-free termination! It's a bit counter-intuitive :-(

Intro to C

```
#include <stdio.h>
/* All time classic hello world */
int main(int argc, char **argv)
{
  printf("Hello World\n");
  return 0;
}
```



Argument count

Arguments: This is a pointer to an array of strings,
we expect 0...argc-1 arguments

Intro to C

```
#include <stdio.h>
/* All time classic hello world */
int main(int argc, char **argv)
{
  int i;
  for(i = 0; i < argc; i++)
    printf("%s\n", argv[i]);
  printf("Hello World\n");
  return 0;
}

/* in mmult we used:
  printf("Program %s\n", argv[argc-1]);
*/
```

Argument count

Argument vector: This is a pointer to an array of strings,
we expect 0...argc-1 arguments

Intro to C

Allows for design of command line programs:
:-) Command line programs are good for building Bioinformatics pipelines!
:-(But are really scary for some Biologists....

GNU utilities for command line parsing: GNU getopt()

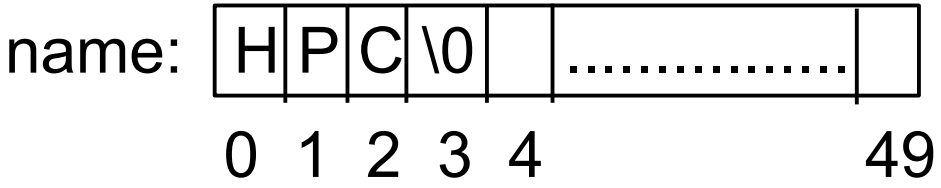
```
#include <stdio.h>
/* All time classic hello world */
int main(int argc, char **argv)
{
  int i;
  for(i = 0; i < argc; i++)
    printf("%s\n", argv[i]);
  printf("Hello World\n");
  return 0;
}

/* in mmult we used:
  printf("Program %s\n", argv[argc-1]);
*/
```

Argument count

Argument vector: This is a pointer to an array of strings,
we expect 0...argc-1 arguments

Strings in C

- Strings in C are represented by arrays of characters!
- The end of the string is marked with a special character, the *null character*, which is simply the character with the value 0 (or \0).
- In some systems char is by default unsigned char and in others not ⟶ problems with portability, unexpected bugs
- For example:
  char name[50] = “HPC”;



Initializing Strings

- char string1[] = "A string declared as an array.\n";
  - We don't use a size declaration for the array; enough memory is allocated for the string, because the compiler knows how long the string constant is. The compiler stores the string constant in the character array and adds a null character (\0) at the end.
- char *string2 = "A string declared as a pointer.\n";
  - This initializations is a pointer to an array of characters. As in the previous example the compiler calculates the size of the array from the string constant and adds a null character. The compiler then assigns a pointer to the first character of the character array to the variable string2.
- char string3[30]; /* usually use multiples of two for init, e.g. 32 here */
- char string[3] = {'H', 'P', 'C'};
  - Declaring a string in this way is useful when you don't know what the string variable will contain, but have a general idea of the length of its contents (in this case, the string can be a maximum of 30 characters long). The drawback is that you will either have to use some kind of string function to assign the variable a value, as the next line of code does (
  - strcpy(string3, "A string constant copied in.\n");
  - or you will have to assign the elements of the array character by character in some kind of loop

String Functions

- C has no built-in facilities for manipulating entire arrays (copying, comparing, etc.)
- Practically no built-in mechanims for manipulating strings:

```
#include <stdio.h>
char firstname[50],lastname[50],fullname[100];
 firstname= "Arnold";              /* Illegal, need to use strcpy here */
 lastname= "Schwarznegger"; /* Illegal, need to use strcpy here */
 fullname= "Mr"+firstname + lastname;
/*     Illegal need to use e.g. strcpy first to copy  “Mr” into fullname and then
        strcat() to concatenate the other two strings in sequence */
```

- The GNU C Library provides some useful functions which handle strings.
- To use the functions beginning with ato (ASCII_to_Something), you must include the header file <stdlib.h> to use the functions beginning with str, you must include the header file <string.h>.

```
int atoi(const char *nptr);       /* ascii to integer */
double atof(const char *nptr); /*ascii to floating point, here actually double */
long atol(const char *nptr);     /* ascii to long integer */
```

String Functions II

- Include **<string.h>** to use the functions below
- **IMPORTANT:** It is the programmer's responsibility to check that the destination array is large enough to hold either what is copied or appended to it.
- **IMPORTANT:** C performs no error checking in this respect.
- The data type size_t is equivalent to unsigned integer or unsigned long integer (machine-dependent!)
- strcpy() copies a string, including the null character terminator from the source string to the destination string. This function returns a pointer to the destination string, or a NULL pointer (check or assert() for errors) on error. The prototype is:

  char *strcpy(char *dst, const char *src);

- strcat() This function appends a source string to the end of a destination string. This function returns a pointer to the destination string, or a NULL pointer on error. Its prototype is:

  char *strcat(char *dst, const char *src);

- strcmp() This function compares two strings.
  - If the first string is greater than the second, it returns a number greater than zero.
  - If the second string is greater, it returns a number less than zero.
  - If the strings are equal, it returns 0. Its prototype is:

    int strcmp(const char *first, const char *second);
  - Checking two strings char *s1, char *s2 for equality means: (strcmp(s1, s2) == 0) needs to be true
- strlen() This function returns the length of a string, not counting the null character at the end. That is, it returns the character count of the string, without the terminator. Its prototype is:

  size_t strlen(const char *str);

String & Text I/O

- Output:
  - printf(“This is a string: %s”, str1);
  - Do a “man printf” there are many options which we can't cover
  - Note that printf() has an argument list of variable size
  - That's a feature of C we won't cover here
- Input:

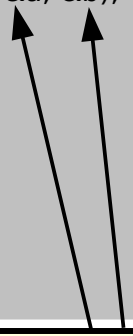
```
#include <stdio.h>
long value1;
double value2;
char str1[1024], str2[1024]; /* long enough ?*/
/* input: “rrr 97533192645 1.2563123487 HPCHPCHPC” */
scanf(“%s%ld%lf%s”, str1, &value1, &value2, str2);
/* A VERY COMMON ERROR: mismatch between %ld and value1 data types ! */
/* the same ons strings: int sscanf(const char *str, const char *format, ...); */
char *str = “rrr 97533192645 1.2563123487 HPCHPCHPC”;
sscanf(str, “%s%ld%lf%s”, str1, &value1, &value2, str2);
/* sscanf returns an integer depending on success/failure of parsing */
```

File I/O

```
#include <stdio.h>
FILE *f;
/* FILE *fopen(const char *path, const char *mode); */
f = fopen("FileName", "w"); /* modes w: write, r: read, a: append */
fprintf(f, "my first C file\n");
fclose(f);
/* It is a very good idea to check if files you want to read actually
exist fopen will return a null pointer otherwise May build a wrapper
function around fopen */

/* reading stuff from a file: */
int a, b;
fscanf(f, "%d %d", &a, &b);
```

File I/O

```
#include <stdio.h>
FILE *f;
/* FILE *fopen(const char *path, const char *mode); */
f = fopen("FileName", "w"); /* modes w: write, r: read, a: append */
fprintf(f, "my first C file\n");
fclose(f);
/* It is a very good idea to check if files you want to read actually
exist fopen will return a null pointer otherwise May build a wrapper
function around fopen */

/* reading stuff from a file: */
int a, b;
fscanf(f, "%d %d", &a, &b);
```



Note the pass by reference here!

Memory Operations

```
#include <stdlib.h>

void *malloc(size_t size);
/* clean alloc: all elements will be set to 0 */
void *calloc(size_t nmemb, size_t size);
/* much nicer than: */
double a = (double*) malloc(sizeof(double) * 100);
for(i = 0; i < 100; i++)
  a[i] = 0.0;

/* free space that has been allocated */
void free(void *ptr);
/* change size of allocated space for a pointer */
void *realloc(void *ptr, size_t size);
/* eg: */
a = (double *)realloc((void *)a, 200);
```

More Memory Operations

```
#include <memory.h> or <string.h>

void *memchr (void *s, int c, size_t n);
/* Search for a character in a buffer */

int memcmp (void *s1, void *s2, size_t n);

/* Compare two buffers, very similar to strcmp !  */

void *memcpy (void *dest, void *src, size_t n) ;

/* Copy one buffer into another, much nicer than doing it explicitly and mostly also more efficient */

double a[100], b[100];

for(i = 0; I < 100; i++)
    a[i] = b[i];

memcpy((void *)a, (void *) b, 100 * sizeof(double));

void *memmove (void *dest, void *src, size_t n);
/* Move a number of bytes from one buffer to another. */

void *memset (void *s, int c, size_t n);
/* Set all bytes of a buffer to a given character. */
```

Advanced Memory Operations

```
#include <sys/mman.h>

int madvise(void *start, size_t length, int advice);

/* The madvise() system call advises the kernel about how to handle paging input/output in the address
       range beginning at address start and with size length bytes.
   It allows an application to tell the kernel how it expects to use some mapped or shared memory areas, so
       that the kernel can choose appropriate  read-ahead  and caching  techniques.   This  call  does not
       influence the semantics of the application (except in the case of MADV_DONTNEED), but may
       influence its performance.
   The kernel is free to ignore the advice. */
/* some advices among others: */
 MADV_RANDOM
   Expect page references in random order.  (Hence, read ahead may be less useful than normally.)
 MADV_SEQUENTIAL
   Expect page references in sequential order.  (Hence, pages in the given range can be aggressively read
       ahead, and  may  be  freed  soon  after  they  are accessed.)
```

Makefiles

Comment line preceded with “#”

```
# Make file for prog executable
prog: input.o prog.o
        cc –o prog input.o prog.o
prog.o: prog.c input.h
        cc –c –o prog.o prog.c
input.o: input.c input.h
        cc –c –o input.o input.c
```

“Dependency line”

“Action line”

Dependency line + Action line = “Rule”

Action line **must** begin with a tab character! Typical Error for Makefiles!

Dependency line **must** start in column 1

Memory Allocation & Variables

Different types consume different amounts of memory! Most architectures store data on "word boundaries", or even multiples of the size (1 byte) of a primitive data type (int, char)

```
char x;
char y='e';
int z = 0x01020304;
```

0x means the constant is interpreted as hex

padding

An int needs 4 bytes

Symbol	Addr	Value
	0	
	1	
	2	
	3	
x	4	?
y	5	'e' (101)
	6	
	7	
z	8	4
	9	3
	10	2
	11	1
	12	

Booleans in C?

- There is no default boolean in C
- We use integers

In C, 0 means “false”, and *any other value* means “true”.

```
int x=4;
int r = 1;
(x < 5)              ⊐ (4 < 5)              ⊐ <true>
(x < 4)              ⊐ (4 < 4)              ⊐ 0
((x < 5) || (x < 4))   ⊐ (<true> || (x < 4))  ⊐ <true>
((x > 5) && (x > 4))   ⊐ (<false> && (x > 5)) ⊐ <false>
((x > 5) && (x > 4))   ⊐ (<false> && (r = (x > 4))) ⊐ <false>
```

Not evaluated because first clause was false

r == 1 due to lazy evaluation :-(

Not evaluated because first clause was true

Booleans: a better way

```
#define FALSE 0
#define TRUE  1
typedef boolean int;

{
  boolean a = FALSE;
  ....
  if(a == TRUE)
      {
       ....
      }
}
```

C PreProcessor

- Three kinds of statements
  - #include
  - #define
  - conditionals (#ifdef, #ifndef, #if, #else, #elif, #endif)

```
#define TRUE 1
#define FALSE 0

#define ABS(x)    (((x)<0)?(-(x)):(x))

/* OS-specific headers usually add -DWIN32 to compiler directives in Makefile*/

#ifdef WIN32
#include <direct.h>
#endif

/* the RAxML -T option, sequential and parallel code in one program, perhaps npt the nicest way to do it
   compiles with -D_USE_PTHREADS
 */

    case 'T':
#ifdef _USE_PTHREADS
          sscanf(optarg,"%d", &NumberOfThreads);
#else
          printf("Option -T does not have any effect with the sequential version.\n");
          printf("It is used to specify the number of threads for the Pthreads-based parallelization\n");
#endif
          break;
```

Operators

== equal to
< less than
<= less than or equal
> greater than
>= greater than or equal
!= not equal
&& logical and
|| logical or
! logical not

The rules of precedence are clearly defined but often difficult to remember or non-intuitive. When in doubt, add parentheses to make it explicit. For oft-confused cases, the compiler will give you a warning "Suggest parens around …" – do it!

+ plus
- minus
*** mult**
/ divide
% modulo

& bitwise and
| bitwise or
^ bitwise xor
~ bitwise not
<< shift left
>> shift right

Don't confuse & and &&..
1 & 2 → 0 *whereas*
1 && 2 → <true>

Using Bit-Vectors
Nice for large presence/absence arrays

```
/* Code from RAxML */
#define BITS_BYTE 8

static const unsigned char bitmask[8] = {1, 2, 4, 8, 16, 32, 64, 128};
/*
 need to play around to figure out which size, i.e., int, long, etc yields best performance
*/


static inline void set_bit(unsigned char *vector, int pos)
{
   vector[pos / BITS_BYTE] |= bitmask[pos % BITS_BYTE];
}

static inline int get_bit(unsigned char *vector, int pos)
{
  if(vector[pos / BITS_BYTE] == 0)
    return 0;
  return ((vector[pos / BITS_BYTE] & bitmask[pos % BITS_BYTE]) == bitmask[pos % BITS_BYTE]);
}
```

Assignments

```
x = y   assign y to x
x++     post-increment x
++x     pre-increment x
x--     post-decrement x
--x     pre-decrement x
```

```
x += y  /* assign (x+y) to x */ x = x + y;
x -= y  /* assign (x-y) to x */ x = x - y;
x *= y  /* assign (x*y) to x */ x = x + y;
x /= y  /* assign (x/y) to x */ x = x / y;
x %= y  /* assign (x%y) to x */ x = x % y;
```

Note the difference between ++x and x++:

```
int x=5;
int y;
y = ++x;
/* x == 6, y == 6 */
```

```
int x=5;
int y;
y = x++;
/* x == 6, y == 5 */
```

Don't confuse = and ==! The compiler will warn "suggest parens".

```
int x=5;
if (x==6)   /* false */
{
  /* ... */
}
/* x is still 5 */
```

```
int x=5;
if (x=6)   /* always true */
{
  /* x is now 6 */
}
/* ... */
```

Assignments

```
x = y   assign y to x
x++     post-increment x
++x     pre-increment x
x--     post-decrement x
--x     pre-decrement x
```

```
x += y  /* assign (x+y) to x */ x = x + y;
x -= y  /* assign (x-y) to x */ x = x - y;
x *= y  /* assign (x*y) to x */ x = x + y;
x /= y  /* assign (x/y) to x */ x = x / y;
x %= y  /* assign (x%y) to x */ x = x % y;
```

Note the difference between ++x and x++:

We can also increment pointers!

```
int x=5;
int y;
y = ++x;
/* x == 6, y == 6 */
```

```
int x=5;
int y;
y = x++;
/* x == 6, y == 5 */
```

Don't confuse = and ==! The compiler will warn "suggest parens".

```
int x=5;
if (x==6)   /* false */
{
  /* ... */
}
/* x is still 5 */
```

```
int x=5;
if (x=6)   /* always true */
{
  /* x is now 6 */
}
/* ... */
```

Pointers

- Pointers have to be declared, just like other variables:

```
int j, k, *ip;
float g, h, *fp;
double m, n, *dp;
char a, b, *cp;
```

- Pointer variable names prefixed by * when declared
- Good practice to give some indication that the variable is a pointer when choosing a name
- Declaration of each pointer reserves an area of storage for holding an address **and an address only!** this address will be the *reference* that the pointer holds to its target object.
- Each pointer variable carries info about the size (byte increment) of the type of object to which it can refer.
- Essentially this means: Each pointer variable knows the value returned by **sizeof** for an object of its target data-type.

Pointer Operations

The address stored as the pointer's reference can be updated using the unary **&** operator eg:

```
int x, y, *ipA *ipB;
x = 15;
ipA = &x;     &x gives ?$*!£~?
ipB = ipA;    ipB's reference is ?$*!£~?
```

where **'?$*!£~?'** is the *meaningless* number which corresponds to the address of integer variable **x.**

ipA and **ipB** both now point to **x.**

Once the pointer's reference has been updated the pointer can be used to manipulate the target address location using the unary *dereferencing* operator * e.g.:

```
y = *ipA;       y holds 15
```

Pointer Ops II

A dereferenced pointer can be used anywhere that the target object could be used:

```
printf("%d", ip);   gives >?$*!£~?
/* whereas */
printf("%d", *ip);  gives >15
```

Pointer variables can be subjected to *pointer arithmetic* eg:

```
int n[10], *ip;
char g[10], *cp;
ip = &n[0];
cp = &g[0];
++ip;   ip holds the address of n[1]
++cp;   cp holds the address of g[1]
```

Note, pointer arithmetic **is consistent with the pointer variable's type (byte increment)**:

so **ip** advances 4 bytes with every **++ip** whereas **cp** advances only one byte for every **++cp** .

Functions

```
int f1(int a, int b)
  {
    int c;
    c = a + b;
    a = 0;
    return c;
  }

void f2(int a, int b, int *c)
  {
   *c = a + b;
  }

void f3(int *a, int b, int *c)
  {
   *c = *a + b;

   *a = 0;
   }

int main(int argc, char *argv[])
{
  int a = 1,
      b = 2,
      c = 0;

  c = f1(a, b);
  f2(a, b, &c);
  f3(&a, b, &c);
}
```

Functions: Passing Arrays

```
void f1(int *a, int length)
  {
    int i;
    for(i = 0; i < length; i++)
      a[i] = 0.0;
  }

void f2(int b[8][8], int length)
  {
    int i, j;
    for(i = 0; i < length; i++)
      for(j = 0; j < length; j++)
        b[i][j] = 0.0;
  }

void f3(int **b, int length)
  {
    int i, j;
    for(i = 0; i < length; i++)
      for(j = 0; j < length; j++)
        b[i][j] = 0.0;
  }

int main(int argc, char *argv[])
{
  int a[8],
      b[8][8],
      i;
  int **bb; /* alloc bb  of course */

  f1(a, 8);
  f2(b, 8);
  f3(bb, 8);
  for(i = 0; i < 8; i++)
    f1(b[i], 8);
    /* or f1(&b[i][0], 8); */
}
```

Recursive Functions

```
int fact(int n)
{
        if(n == 0)
          return  1;
        else /* don't really need else here */
          return n * fact(n - 1);
}

int fact(int n)
{
  return (n == 0 ? 1: n * fact(n - 1));  /* a nasty C shortcut */
}
/* REMEMBER: doing this iteratively is better */
int factIterative(int n)
{
 int res = 1;
 int i;
 for(i = 1; i < n; i++)
   res *= res;

 return res;
}
```

Structs

- Group several pieces of related information together and is a collection of variables, referred to as fields, under a single name.
- Fields can be of differing types and must be uniquely named within the structure
- As the storage needed cannot be known a priori by the compiler, a definition is initially required.
- A structure can then be further defined as a new named type thus extending the number of available data types.
- It can use other structures, arrays or pointers as some of its members
- Uses the keyword **struct**
- **We can of course also define arrays of structs or new data types!**

Structs

```
/* Structures can also contain structures. */

struct date
{
  int  month, day, year;
};

struct time
{
  int  hours, mins, secs;
};

struct date_time
{
  struct date sdate;
  struct time stime;
};
```

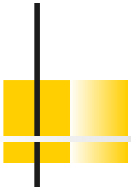
Structs

- Each field can be separately assigned/initialized
- Assignments are analogous to the standard for simple variables of the specific type

```
st_record.age = 19;
st_record.year = 2000;
```

- `st_record.year` will behave just like a normal integer.
- However, it is referred to by st_record.name
- Here the dot is an operator that selects a field from a structure.

In a way Structs are the Predecessors of Objects



- Introduction of objects to better handle structs (or associated typedefs) is a natural extension
- There are no functions to assign, initialize, copy, free the memory required by structs in an oo-fashion
- Need to do this manually!
- However a lot of this can be automated!
- structs can be passed by reference or by passed by value !

Recursive Structs

```
struct node
{
  int value;
  struct node *next;
};
typedef struct node *Listpointer;  /* Listpointer is a new type we defined */
/* alternatively : */

typedef struct node
{
  int value;
  struct node *next;
} *Listpointer;
void print_list(Listpointer ptr)
 {
   while (ptr != NULL)
     {
       printf("%d", ptr -> value);/ * print value */
       ptr = ptr -> next;          // point to next
     }
}
```

Dynamic Arrays & Structs as Pointers

```
typedef struct dynArray
{
  size_t entries;
  size_t max;
  int *a;
} dynArrayStruct, *dynArrayPtr;

dynArrayPtr da = (dynArrayPtr)malloc(sizeof(dynArrayStruct));
da->entries = 0;
da->max = 100;
da->a = (int *)malloc(sizeof(int) * da->max);

void addValue(int value, dynArrayPtr da)
 {
   if(da->entries < da->max)
         {
       da->a[da->entries++] = value;
       return;
     }

    if(da->entries == da->max)
          {
                int *buf = (int*)malloc(sizeof(int) * 2 * da->max);
          memcpy((void *)buf, (void *)(da->a), sizeof(int) * da->max);
          free(da->a);
                da->a = buf;
                da->max *= 2;
                da->a[da->entries++] = value;
          }
 }
```

Pointers to Functions

- Perhaps not always a good idea, we had some problems on the IBM BlueGene/L supercomputer
  - not all compilers like this :-(
- Allows for generic programming style, e.g., generic quicksort:

```
#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size, int(*compar)(const void *, const void *));

/* an example: */

double a[100];
....

int doubleComparer(const void *x, const void *y)
 {
            double *_xp = (double *)x;
            double *_yp = (double *)y;

            double _x = *_xp;
            double _y = *_yp;

            if(_x == _y)
              return 0;
            if(x > y)
       return 1;
            else
              return -1; /* I honestly never remeber which order ascending or
                            descending this will induce .... trial and error :-( */
 }


qsort((void *)a, 100, sizeof(double), doubleComparer);
```

Less readable

```
#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size, int(*compar)(const void *, const void *));

/* an example: */

double a[100];
....

int doubleComparer(const void *x, const void *y)
  {
              double _x = *((double *)x); / put parentheses here for better readability */
              double _y = *((double *)y);

              if(_x == _y)
                return 0;
              if(x > y)
        return 1;
              else
                return -1; /* I honestly never remeber which order ascending or
                                    descending this will induce .... trial and error :-( */
  }


qsort((void *)a, 100, sizeof(double), doubleComparer);
```

Good Programming Practice

- Use assertions whenever you can
- Always use default clause in case switches
- Don't forget break statements for every case program will execute code for next state

```
#define STATE_0 0
#define STATE_1 1

int state;

switch(state)
  {
  case STATE_0:
        ...
        break;
  case STATE_1:
     ...
     break;
  default:
        assert(0);
  }
```

```
#define STATE_0 0
#define STATE_1 1

int state;

switch(state)
  {
  case STATE_0:
        ...
  case STATE_1:
     ...
     break;
  default:
        assert(0);
  }
```

Good Programming Practice

- Every function should be preceded by a comment describing the purpose of the function.
- The last character printed by a function that displays output should be a newline (\n). This ensures that the function will leave the screen cursor positioned at the beginning of a new line. Conventions of this nature encourage software reusability—a key goal in software development environments.
- Choosing meaningful variable names helps make a program self-documenting, i.e., fewer comments are needed.
- The first letter of an identifier used as a simple variable name should be a lowercase letter.
- Separate “words” within identifiers with underscores or mixed upper and lower case.
- Examples: surfaceArea surface_Area surface_area
- Use all uppercase for **symbolic constants** (used in **#define** preprocessor directives).
- Note: symbolic constants are not variables, but make the program easier to read.
- Examples
- #define PI 3.14159
- #define AGE 52

Good Programming Practice

- Do not “hide” the initialization
  - put initialized variables on a separate line
  - a comment is always a good idea
  - Example:
  - int height ; /* rectangle height */
  - int width = 6 ; /* rectangle width */
  - int area ; /* rectangle area */
  - NOT int height, width = 6, area ;
- Place each variable declaration on its own line !
- Place a comment before each logical “chunk” of code to describe what it does.
- Do not place a comment on the same line as code (with the exception of variable declarations).
- **Use spaces around all arithmetic and assignment operators.**
- Use blank lines to enhance readability.
- Comments should explain why you are doing something, not what you are doing it.
  a = a + 1 /* add one to a */ /* WRONG */
  /* count new student */ /* RIGHT*/

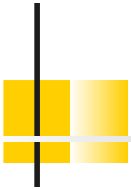
Common Errors & Problems

- Example: Find the average of three variables a, b and c
  - Do not use: a + b + c / 3 == (a + b) + (c / 3)
  - Use: (a + b + c) / 3
- Using = instead of ==; a = b is always true!
- “;” after an if statemenet if(a == b); while there is still code to come below
  - Integer division truncates remainder (floor function)
    - 7 / 5 evaluates to 1 (int a = ⌊7/5⌋) !
- Mentioned before: beware of char being interpreted as signed char versus unsigned char
- Beware of size_t in malloc which can get much larger than a standard signed int!
- Parsing input files: carriage returns in Windows, MAC, and Linux/Unix systems (sometimes it's '\n' Linux/Unix and sometimes '\r' :-( )

Dangling Else

```
char y;
int x;
if(x > 0)
  if(y == 'A')
     printf(“Positive A”);
   else /* to which if does this else refer to */
     printf(“Negative”);
/* add parentheses if in doubt, I always am! This is much better programming style */

if(x>0)
  {
    if(y==’A’)
      printf(“Positive A”);
   }
  else
    printf(“Negative”);
```

Learning how to program



- You will not learn anything if you do not start experimenting and gaining experience yourself
- Most of the programming recommendations stem from painful experiences
- Don't be as stupid as your instructor
- ... and think that you are a good programmer and don't need all this!