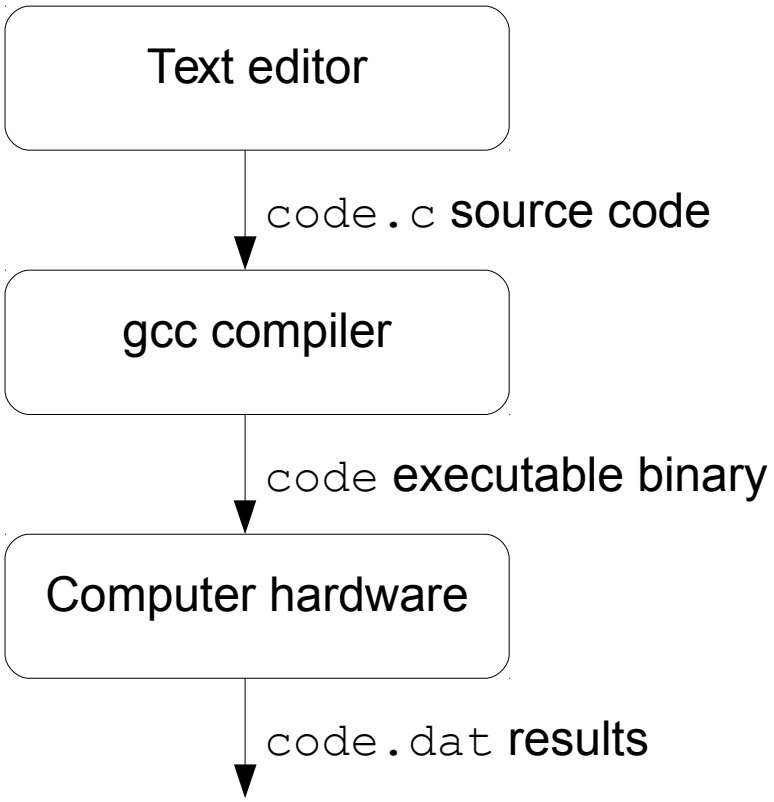
Class Progress

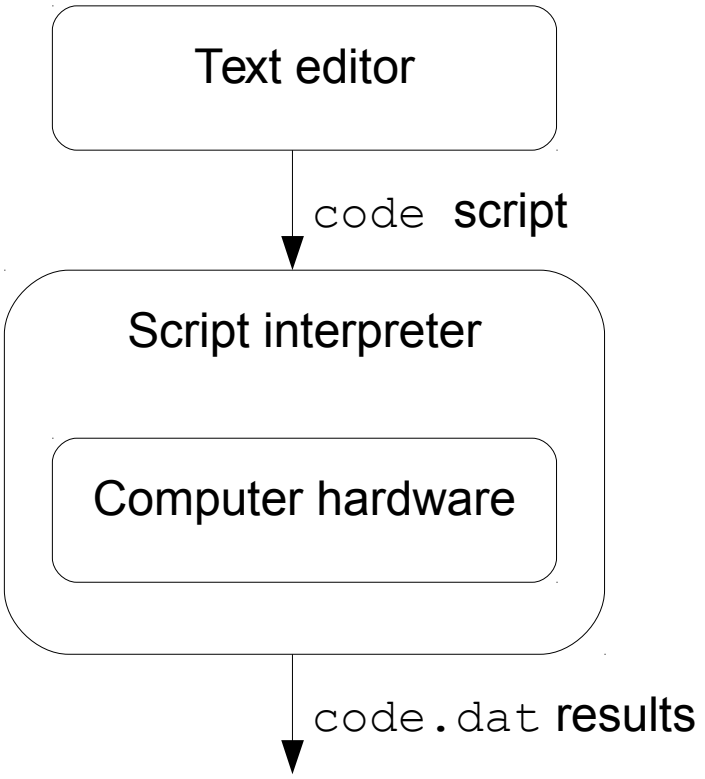
Basics of Linux, gnuplot, C
Visualization of numerical data
Roots of nonlinear equations
(Midterm 1)
Solutions of systems of linear equations
Solutions of systems of nonlinear equations
Monte Carlo simulation
Interpolation of sparse data points
Numerical integration
(Midterm 2)
Solutions of ordinary differential equations

Language Choices for Numerical Analysis

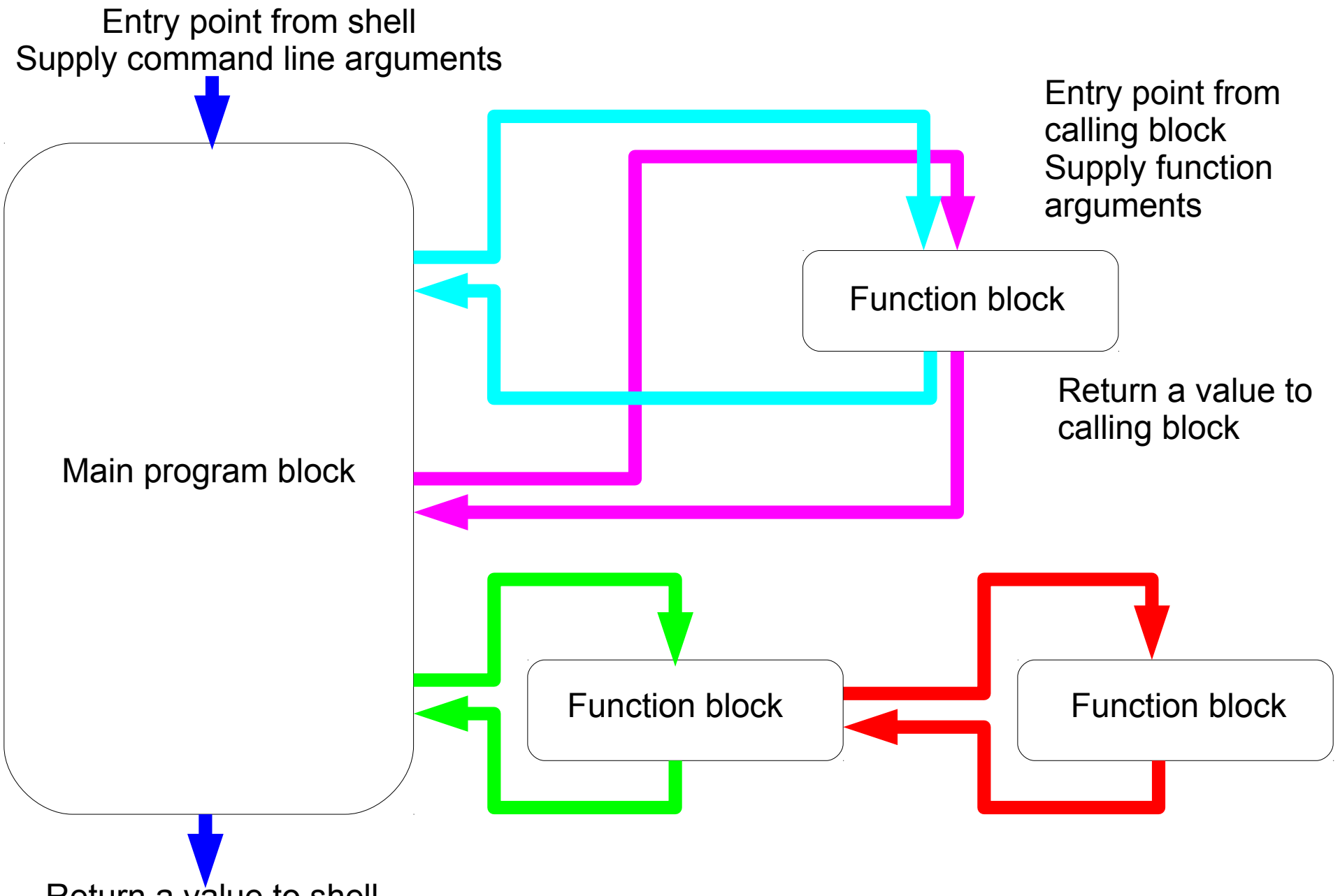
Compiled languages
(C, Fortran)



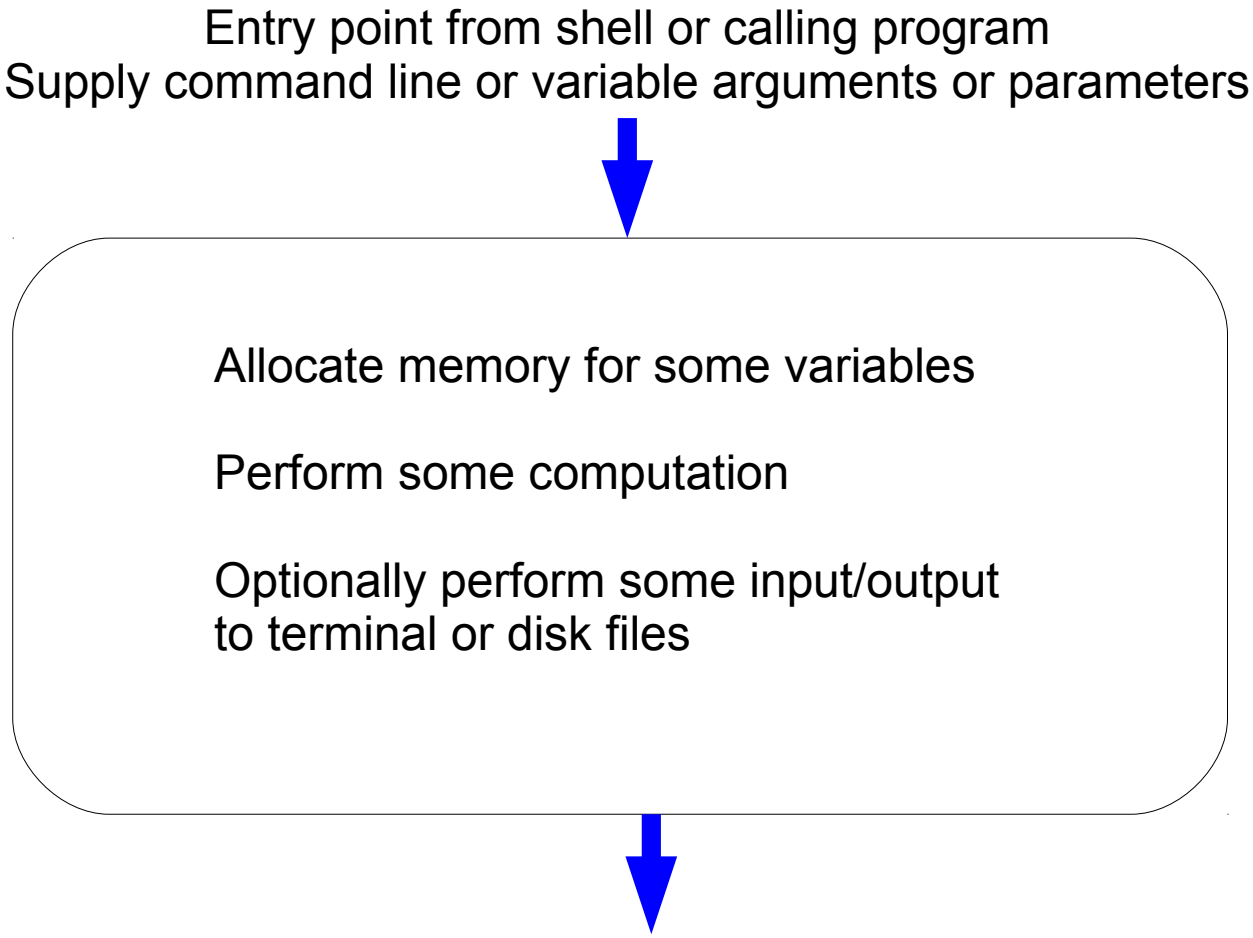
Interpreted languages
(perl, tcl, python, etc)



Program Structure



Program Blocks



Defining C Main Program Blocks

Main programs must be named 'main'

Main programs are started with two arguments assembled from the shell command line, an integer argument count and an array of pointers to character strings

```
int main(int argc,char *argv[]) {

   ...
   ...
   ...

   exit(0);
}
```

Main programs must return an integer exit code to the shell

Program statements are listed within the curly braces

A call to the `exit` function terminates the main program and returns its argument as the exit code to the shell

There may be more than one exit statement in the code for a program, within conditional statements

Defining C Function Blocks

Any desired number of arguments of any type, named with their 'dummy' names, used locally within the function

Functions may be named any desired unique name

```
double solve(int n,double x,double y) {

   ...
   ...
   ...

   return(alpha + 89.0);
}
```

Functions may return a value any desired type to the calling program

Function statements are listed within the curly braces

A call to the `return` function terminates the function and returns its argument to the calling main program or function

There may be more than one return statement in the code for a function, within conditional statements

Constants in C

Integer:

```
3
17
123456
```

Floating point: (includes a decimal point or an 'e' exponent):

```
1.23
0.056789
6.02e23
1.60e-19
```

(means 6.02×10^{23})

(means 1.60×10^{-19})

Character (enclose in single quotes):

```
'a'
' '
'\n'
```

String (type pointer to char, enclose in double quotes):

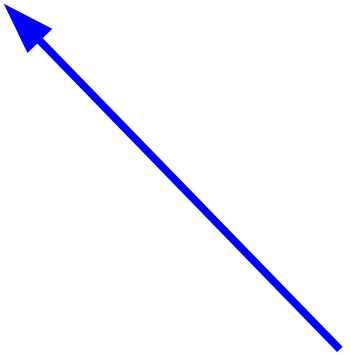
```
"This is a line of text"
"Line one\nLine two\nLine three"
```

Statement Types in C for Numerical Programs

- Declaration statements
- Arithmetic assignment statements
- Compound statements
- Conditional statements
- Iterative statements

Declaration Statements

```
type variable,variable,...,variable;
```



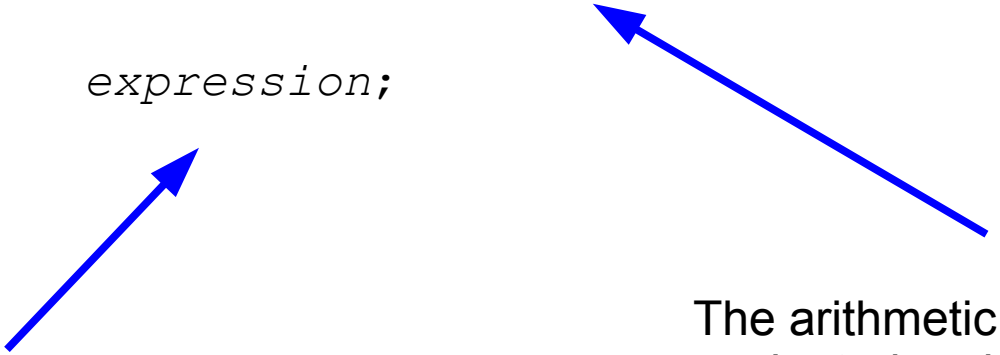
Each variable name is declared to be of the specified type, and space in memory is allocated to hold its value.

Examples of Declaration Statements

```
int i,j,k;
double x,y,a;
char c;
int *i_ptr;
double *d_ptr;
```

Arithmetic Assignment Statements

```
variable = expression;

expression;
```



The arithmetic expression is evaluated and the resulting value is discarded. This is useful when the expression evaluation has some side effect, like some function being called or a variable incremented.

The arithmetic expression is evaluated and the resulting value is assigned to the variable

The = sign is equivalent to the ← assignment operator in Cheney and Kincaid style pseudocode

Examples of Assignment Statements

```
i = 0;
j = (i + 3) * 8;
x = 3.14159265358979323846264338327950288;
y = (sqrt(x) + 57.0) / 7.89;
c = 'a';
i++;
putchar('*');
exit(0);
```

Order of Precedence of Arithmetic Operators

(high)	`( ) [ ]`	grouping, subscripting
	`++ -- + - * &`	unary increment, decrement, sign, pointer dereference, address of
	`* / %`	multiply, divide, modulo
	`+ -`	add, subtract
	`< <= > >=`	comparison test
	`== !=`	equality, inequality test
(low)	`=`	assignment

Operator Precedence in Assignment Statements

Use parentheses in C code to get the arithmetic operations you really want!

```
j = i + 3 * k;
    or
j = i + (3 * k);
```

$$j \leftarrow i + 3\mathrm{k}$$

```
j = (i + 3) * k;
```

$$j \leftarrow (i + 3)k$$

```
y = (sqrt(x) + 57.0) / radius;
```

$$y \leftarrow \frac{\sqrt{x} + 57.0}{r}$$

```
y = sqrt(x) + 57.0 / radius;
    or
y = sqrt(x) + (57.0 / radius);
```

$$y \leftarrow \sqrt{x} + \frac{57.0}{r}$$

Operator Associativity in Assignment Statements

Use parentheses in C code to get the arithmetic operations you really want!

```
a = x - y + z;
    or
a = (x - y) + z;
```

$$a \leftarrow (x - y) + z$$

```
a = x - (y + z);
```

$$a \leftarrow x - (y + z)$$

```
a = x / y * z;
    or
a = (x / y) * z;
```

$$a \leftarrow \frac{x}{y} \cdot z$$

```
a = x / (y * z);
```

$$a \leftarrow \frac{x}{y \cdot z}$$

Compound Statements

```
{
   statement;
   statement;
   statement;
     .
     .
     .
   statement;
   statement;
}
```

Any number of simple statements of any type, enclosed in curly braces { } form a compound statement and may be used in any place a single statement is expected, such as a part of conditional or iterative statements.

Note: each component statement is terminated with a semicolon ';' but the overall compound statement is not.

Conditional Statements

```
if (expression) statement;
else statement;
```

```
if (expression) {
  statement;
  statement;
}
```

```
if (expression) {
  statement;
  statement;
}
else {
  statement;
  statement;
}
```

```
if (expression) {
  statement;
  statement;
}
else if (expression) {
  statement;
  statement;
}
else if (expression) {
  statement;
  statement;
}
  .
  .
  .
else if (expression) {
  statement;
  statement;
}
else {
  statement;
  statement;
}
```

Comparison Operators in Conditional Expressions

`<`	`less than`
`<=`	`less than or equal to`
`>`	`greater than`
`>=`	`greater than or equal to`
`==`	`equality`
`!=`	`inequality`

Never use `==` or `!=` in comparison expressions with floating point numbers!
The comparison can be invalid because of rounding in the least significant few bits!
`==` and `!=` are for comparing int values only

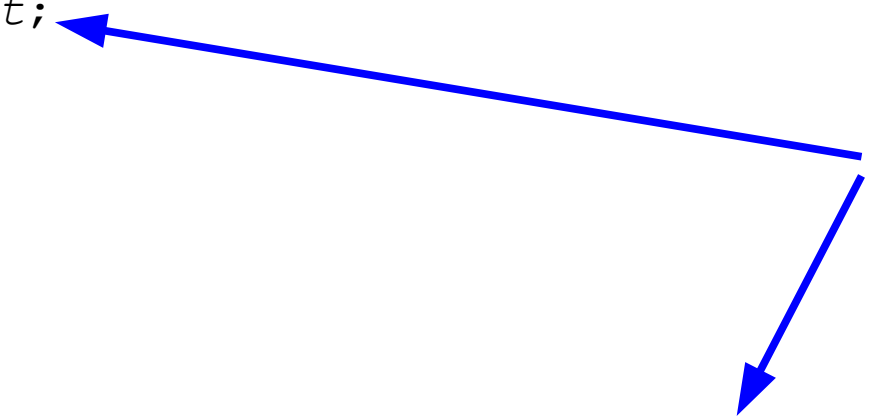
Examples of Conditional Statements

```
if (energy > 3.0 * k * T) {
  printf("High energy particle detected\n");
}
else if (energy < k * T) {
  printf("Low energy particle detected\n");
}
else {
  particles++;
}
```

Note indentation and consistent placement of { and }

Iterative Statements

```
while (expression) statement;

while (expression) {
  statement;
    ...
  statement;
}

for (init_statement; expression; incr_statement) statement;

for (init_statement; expression; incr_statement) {
  statement;
    ...
  statement;
}

do {
  statement;
  statement;
} while (expression);
```



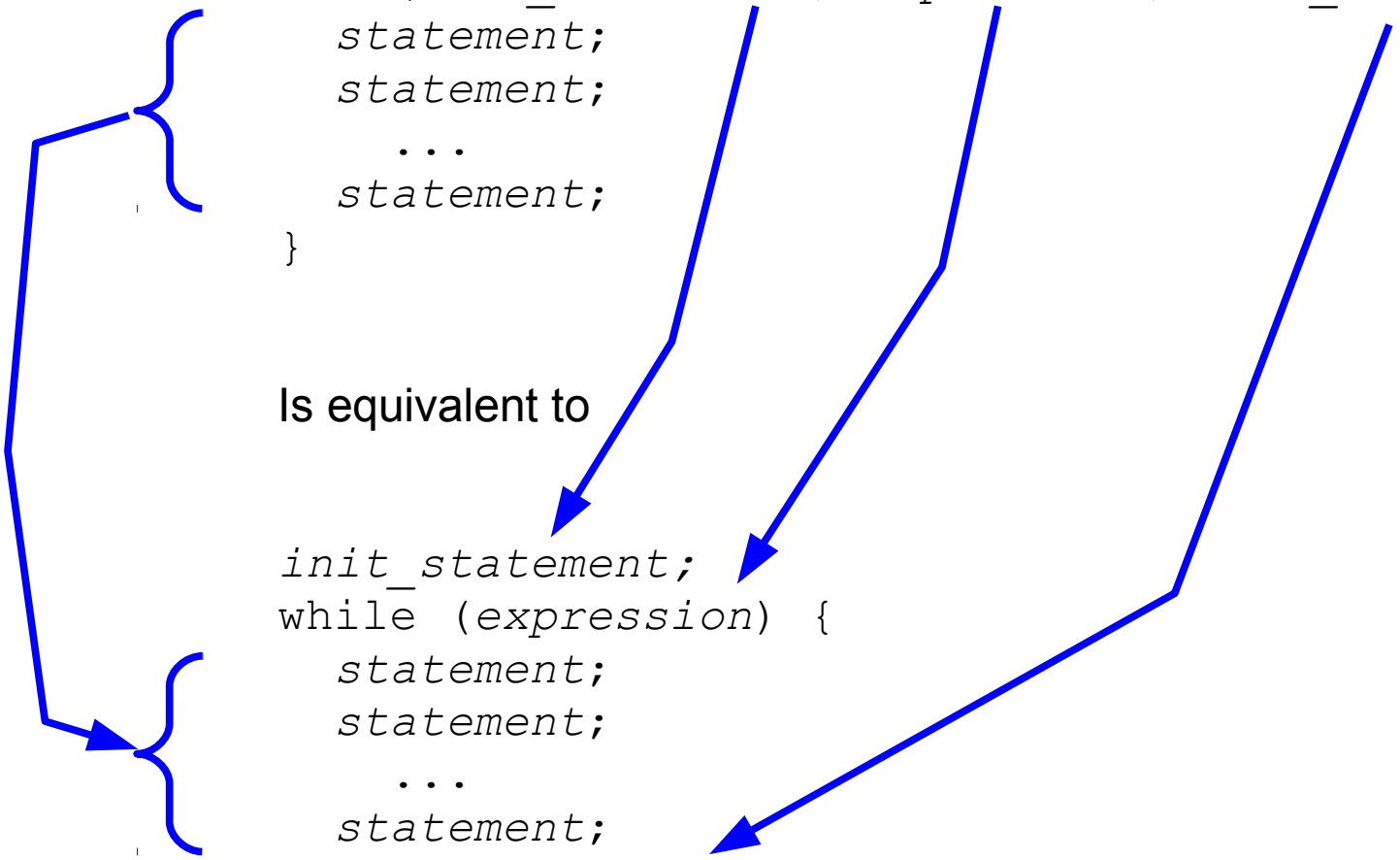
Simple statements may be expanded to a compound statement

Equivalence of `while` and `for` Statements

```
for (init_statement; expression; incr_statement) {
  statement;
  statement;
    ...
  statement;
}
```

Is equivalent to

```
init_statement;
while (expression) {
  statement;
  statement;
    ...
  statement;
  incr_statement;
}
```



Examples of Iterative Statements

```
i = 0;
while (i < 10) {
  velocity[i] = 0.0;
  i++;
}

for (i = 0; i < 10; i++) {
  velocity[i] = 0.0;
}

i = 0;
while (i < 10) velocity[i++] = 0.0;

for (i = 0; i < 10; i++) velocity[i] = 0.0;

i = 0;
do {
  velocity[i] = 0.0;
  i++;
} while (i < 10);
```

Note indentation and consistent placement of { and }

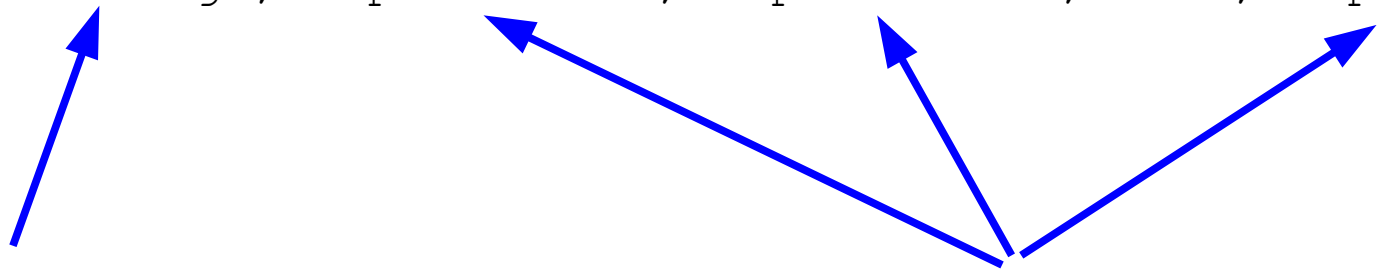
Examples of “Infinite Loops”

```
i = 0;
while (i < 10) {
  velocity[i] = 0.0;
}

i = 0;
while (i < 10) velocity[i] = 0.0;

i = 0;
do {
  velocity[i] = 0.0;
} while (i < 10);
```

These constructions will result in your program getting stuck for an arbitrarily long time. Hit 'Control-c' at the terminal window to interrupt and terminate.

printf() Library Function

```
printf(format string , expression1 , expression2 , ... , expressionN);
```



Character string that provides both literal characters to be included in the output as well as N place holders describing how the numerical values are to be translated to text and inserted into the output, in sequence 1 through N.

N numerical values to be output, where N may be 0 to any number

printf() Library Function Examples

Assume:

```
double x,y;
int i,j;
x = sqrt(2.0);
y = 3.14159265358979;
i = 7;
j = -39;
```

```
printf("The values are %d, %d, %f, %f\n",i,j,x,y);
```

would generate into standard output

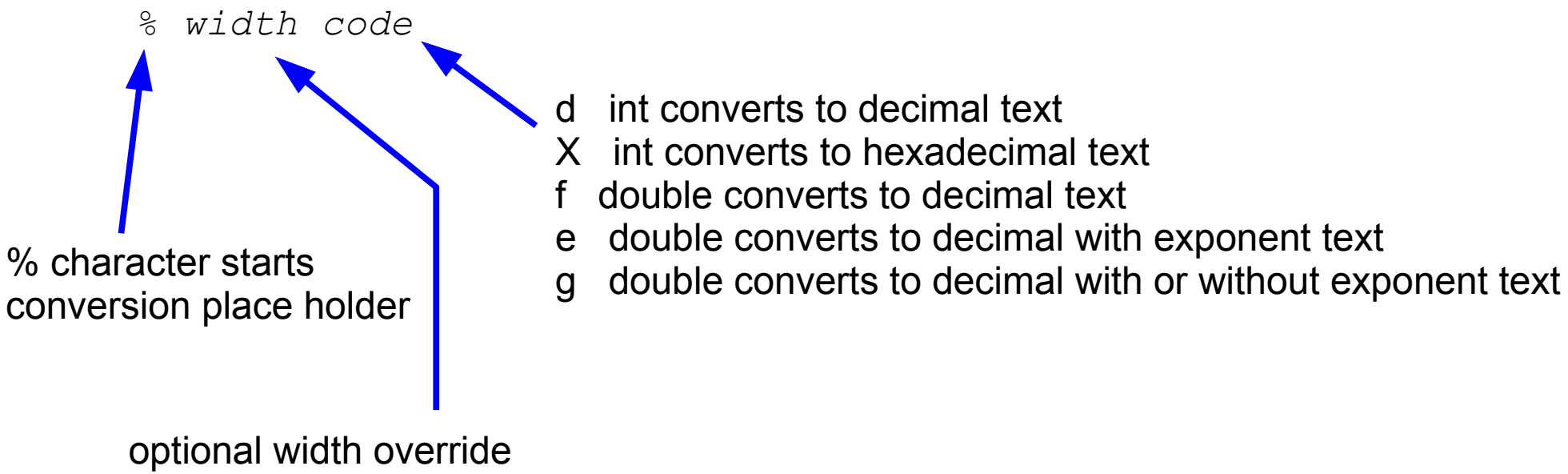
```
The values are 7, -39, 1.414214, 3.141593
```

```
printf("The element at (%d,%d) is:\n  x=%.8f y=%.8f\n",i,j,x,y);
```

would generate into standard output

```
The element at (7,-39) is:
  x=1.41421356 y=3.14159265
```

printf() Library Function Conversion Codes



Beware of Mixing Data Types in Expressions!

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

int main(int argc,char *argv[]) {
  int i,j,k,m;
  double x,y,z;

  i = 7;
  j = 4;
  k = i / j;
  m = i % j;
  printf("k=%d m=%d\n",k,m);
  i = 1;
  j = 10;
  k = i / j;
  m = i % j;
  printf("k=%d m=%d\n",k,m);
  x = 1.0;
  y = x + 1.0 / 10.0;
  z = x + 1 / 10;
  printf("y=%.2f z=%.2f\n",y,z);
  x = 1.0;
  y = x + 0.1;
  z = x + i / j;
  printf("y=%.2f z=%.2f\n",y,z);
  exit(0);
}
```

Integer division results in whole number quotients and remainders

Mixing constants or expressions of integer type into floating point expressions can lead to numerical errors!

Output:

```
k=1 m=3
k=0 m=1
y=1.10 z=1.00
y=1.10 z=1.00
```

Intentionally Mixing Data Types

Example: Compute mean of floating point values:

```
int i,n;
double sum,mean;

n = ...;

i = 0;
sum = 0.0;
while (i < n) {
   sum = sum + ...;
   i++;
}
mean = sum / ((double) n);
```

Integer variable holds count of values for loop control

Option 1: Use “cast” operator to cast argument as a different type in a documented way
integer n is cast as a double to match type of division expression

Intentionally Mixing Data Types

Example: Compute mean of floating point values:

```
int i,n;
double sum,mean,n_double;

n = ...;
n_double = n;

i = 0;
sum = 0.0;
while (i < n) {
   sum = sum + ...;
   i++;
}
mean = sum / n_double;
```

Integer variable holds count of values for loop control

Option 2: Use assignment statement to convert type across assignment operator
n_double is already of type double to match type of division expression

Be Careful Converting int to double!

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

int main(int argc,char *argv[]) {
  double x,y,z;
  int i,j;

  x = 10.0;
  y = 1.0e-18;
  z = x - y;  i = z;  j = z + 0.5;
  printf("y=%.3g z=%20.16f i=%2d j=%2d\n",y,z,i,j);
  y = 1.0e-17;
  z = x - y;  i = z;  j = z + 0.5;
  printf("y=%.3g z=%20.16f i=%2d j=%2d\n",y,z,i,j);
  y = 1.0e-16;
  z = x - y;  i = z;  j = z + 0.5;
  printf("y=%.3g z=%20.16f i=%2d j=%2d\n",y,z,i,j);
  y = 1.0e-15;
  z = x - y;  i = z;  j = z + 0.5;
  printf("y=%.3g z=%20.16f i=%2d j=%2d\n",y,z,i,j);
  y = 1.0e-14;
  z = x - y;  i = z;  j = z + 0.5;
  printf("y=%.3g z=%20.16f i=%2d j=%2d\n",y,z,i,j);
  exit(0);
}
```

Conversion of double to int across assignment operator gives a chop operation, not a round! Explicitly add 0.5 before conversion to get a round operation!

Output:

```
y=1e-18 z= 10.0000000000000000 i=10 j=10
y=1e-17 z= 10.0000000000000000 i=10 j=10
y=1e-16 z= 10.0000000000000000 i=10 j=10
y=1e-15 z=  9.9999999999999982 i= 9 j=10
y=1e-14 z=  9.9999999999999893 i= 9 j=10
```

Intentionally Mixing Data Types

Example: For a histogram, increment bin count corresponding to a floating point value:

```
int count[10];
double bin,value;

bin = value / 10.0;

count[(int) (bin + 0.5)]++;
```

Floating point variable holds a bin index corresponding to a floating point value

Option 1: Use “cast” operator to cast argument as a different type in a documented way
double bin is cast as an int to use as an array index
note adding 0.5 to turn conversion to int, by default a 'floor' operation, into a round operation

Intentionally Mixing Data Types

Example: For a histogram, increment bin count corresponding to a floating point value:

```
int i_bin,count[10];
double bin,value;

bin = value / 10.0;

i_bin = bin + 0.5;
count[i_bin]++;
```

Floating point variable holds a bin index corresponding to a floating point value

Option 2: Use assignment statement to convert type across assignment operator
double bin is cast as an int to use as an array index
note adding 0.5 to turn conversion to int, by default a 'floor' operation, into a round operation

Program 1 to Plot Damped Oscillator Response

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

int main(int argc,char *argv[]) {
  double t,disp,w0,Q;

  w0 = 6.28318530718;
  Q = 10.0;
  t = 0.0;
  while (t < 10.001) {
    disp = exp(-0.5 * w0 * t / Q) * cos(w0 * t);
    printf("%.8g %.8g\n",t,disp);
    t = t + 0.01;
  }
  exit(0);
}
```

Standard header files in /usr/include
Declare function and data types for using functions in C standard library

Start of main program block

Declare variables

Initialize some variables

Iterative loop using '`while`' steps over time

Compute amplitude as a function of time

Send time and amplitude data points to standard output

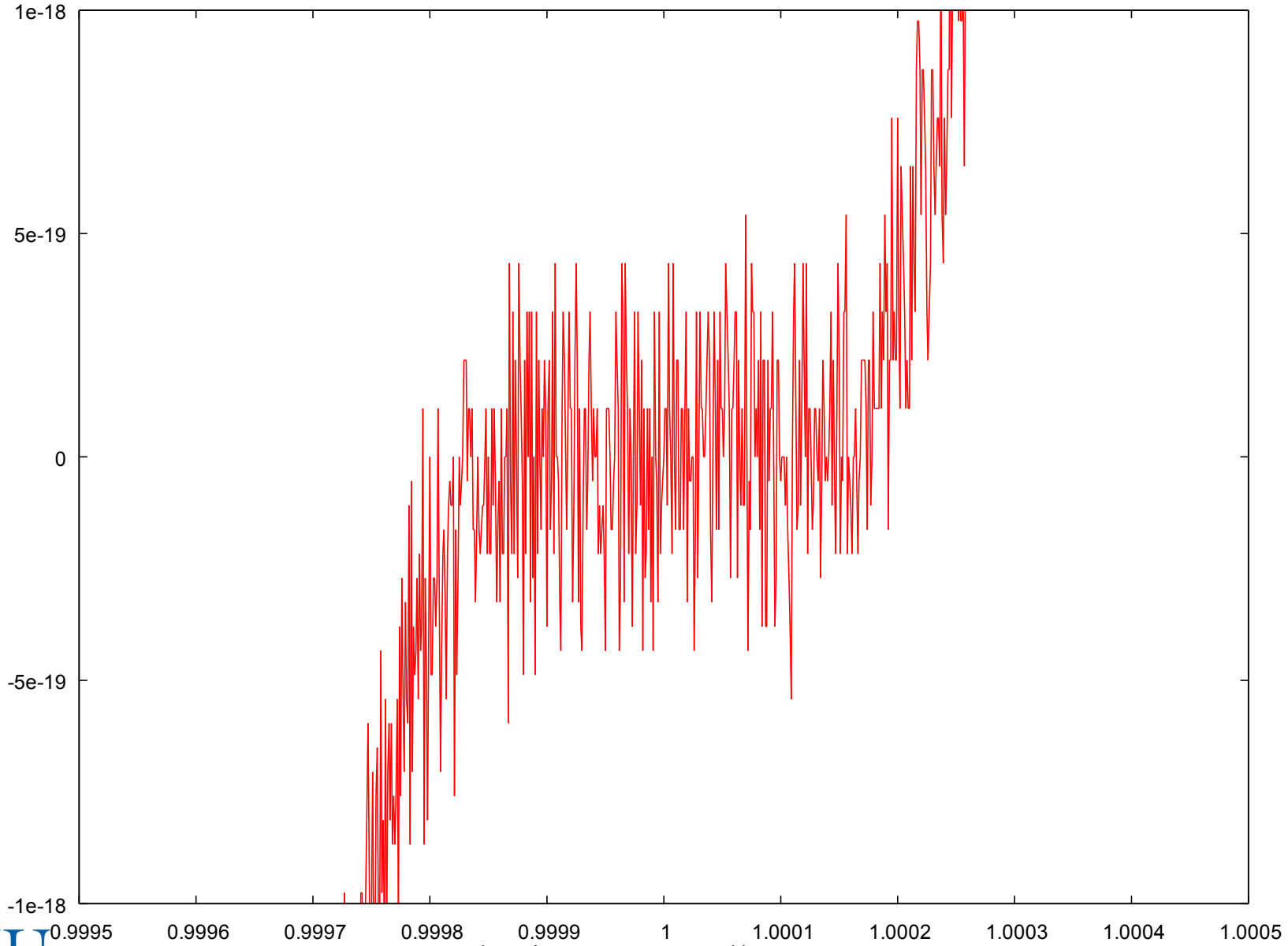
Exit main program

Program 2 to Plot Damped Oscillator Response

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

int main(int argc,char *argv[]) {
  double t,disp,vel,w0,Q;

  w0 = 6.28318530718;
  Q = 10.0;
  t = 0.0;
  while (t < 10.001) {
    disp = exp(-0.5 * w0 * t / Q) * cos(w0 * t);
    vel = -exp(-0.5 * w0 * t / Q) * w0 * sin(w0 * t);
    printf("%.8g %.8g %.8g\n",t,disp,vel);
    t = t + 0.01;
  }
  exit(0);
}
```

Using Floating Point Variables with Comparison Operators

Remember this plot from the lecture on numerical roundoff errors?



Using Floating Point Variables with Comparison Operators

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

int main(int argc,char *argv[]) {
  int i;
  double x;

  i = 0;
  x = 0.0;
  while (i < 10) {
    x = x + 0.7;
    printf("x=%.16f\n",x);
    i++;
  }
  if (x < 7.0) printf("x is less than 7.0\n");
  else if (x == 7.0) printf("x is equal to 7.0\n");
  else printf("x is greater than 7.0\n");
  exit(0);
}
```

Output:

```
x=0.7000000000000000
x=1.3999999999999999
x=2.0999999999999996
x=2.7999999999999998
x=3.5000000000000000
x=4.2000000000000002
x=4.9000000000000004
x=5.6000000000000005
x=6.3000000000000007
x=7.0000000000000009
x is greater than 7.0
```

Loop adds 0.7 ten times

The expected value is 7.0

Using Floating Point Variables with Comparison Operators

The C '>' greater than test will involve **all 64 bits** of a double precision variable with the exactness of digital logic:

1.0000000000000002 =

0 01111111111 1. 0000 00000000 00000000 00000000 00000000 00000000 00000001

Is greater than

1.0000000000000000 =

0 01111111111 1. 0000 00000000 00000000 00000000 00000000 00000000 00000000

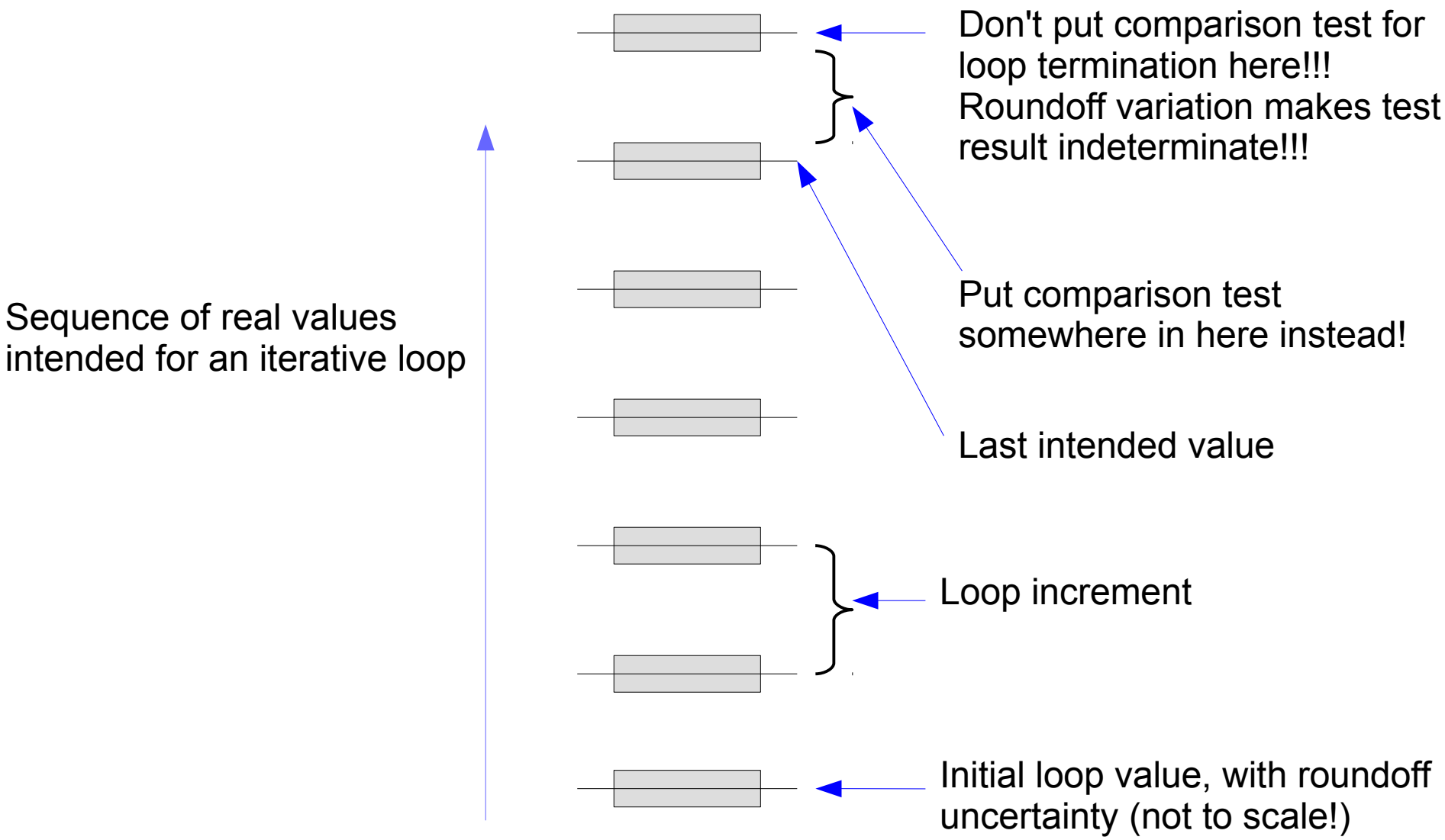
Is greater than

0.9999999999999989=

0 01111111110 1. 1111 11111111 11111111 11111111 11111111 11111111 11111111

The C '==' equality test will not report equality unless **all 64 bits** of double precision variables exactly match!

Using Floating Point Variables with Comparison Operators



Using Floating Point Variables with Comparison Operators

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

int main(int argc,char *argv[]) {
  int i;
  double x;

  i = 0;
  x = 0.0;
  while (x < 1.0) {
    printf("i = %d x=%.16f\n",i,x);
    i++;
    x = x + 0.1;
  }
  exit(0);
}
```

Output:

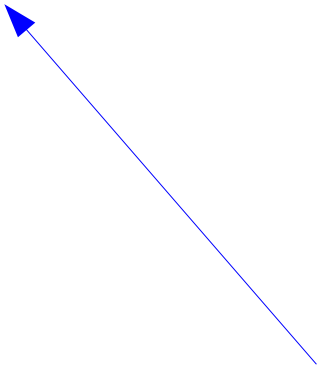
```
i = 0 x=0.0000000000000000
i = 1 x=0.1000000000000000
i = 2 x=0.2000000000000000
i = 3 x=0.3000000000000000
i = 4 x=0.4000000000000000
i = 5 x=0.5000000000000000
i = 6 x=0.6000000000000000
i = 7 x=0.7000000000000000
i = 8 x=0.7999999999999999
i = 9 x=0.8999999999999999
i = 10 x=0.9999999999999999
```

The loop is executed 11 times instead of the expected 10 times

Changing the expression to test for `(x < 0.999)` would be an easy fix

Using Floating Point Variables with Comparison Operators

```
double x,xstart,xstop,xinc;

   ...
xstart = ...;
xstop = ...;
xinc = ...;
   ...
   ...
xstop = xstop + 0.5 * xinc;
x = xstart;
while (x < xstop) {
   ...
   x = x + xinc;
}
```



Example of changing the value to test for loop termination, in this case to allow one final iteration at the stop value

C Data Types for Numerical Programming

- Pointer
  - Named "*any data type* *", for instance "int *", "double *", "char *"
  - Usually 32 bits, 4 bytes
- Arrays
  - Multiple sequential instances of any data type
- User defined structures
  - Named "struct *name*", for instance "struct point2d"
  - May be any collection of other data types, including arrays or other structures

Regular and Pointer Data Types

A plain symbol refers to the value of a variable, examples:

```
double x,a;
int i,j;
```

Preceding the symbol with the '&' character in an expression generates the address of that variable as a pointer constant:

```
&x
&j
```

Preceding a pointer variable with the '*' character "dereferences" the pointer back to the value it points to:

```
char c,d,*ptr;

ptr = &d;
c = *ptr;
```

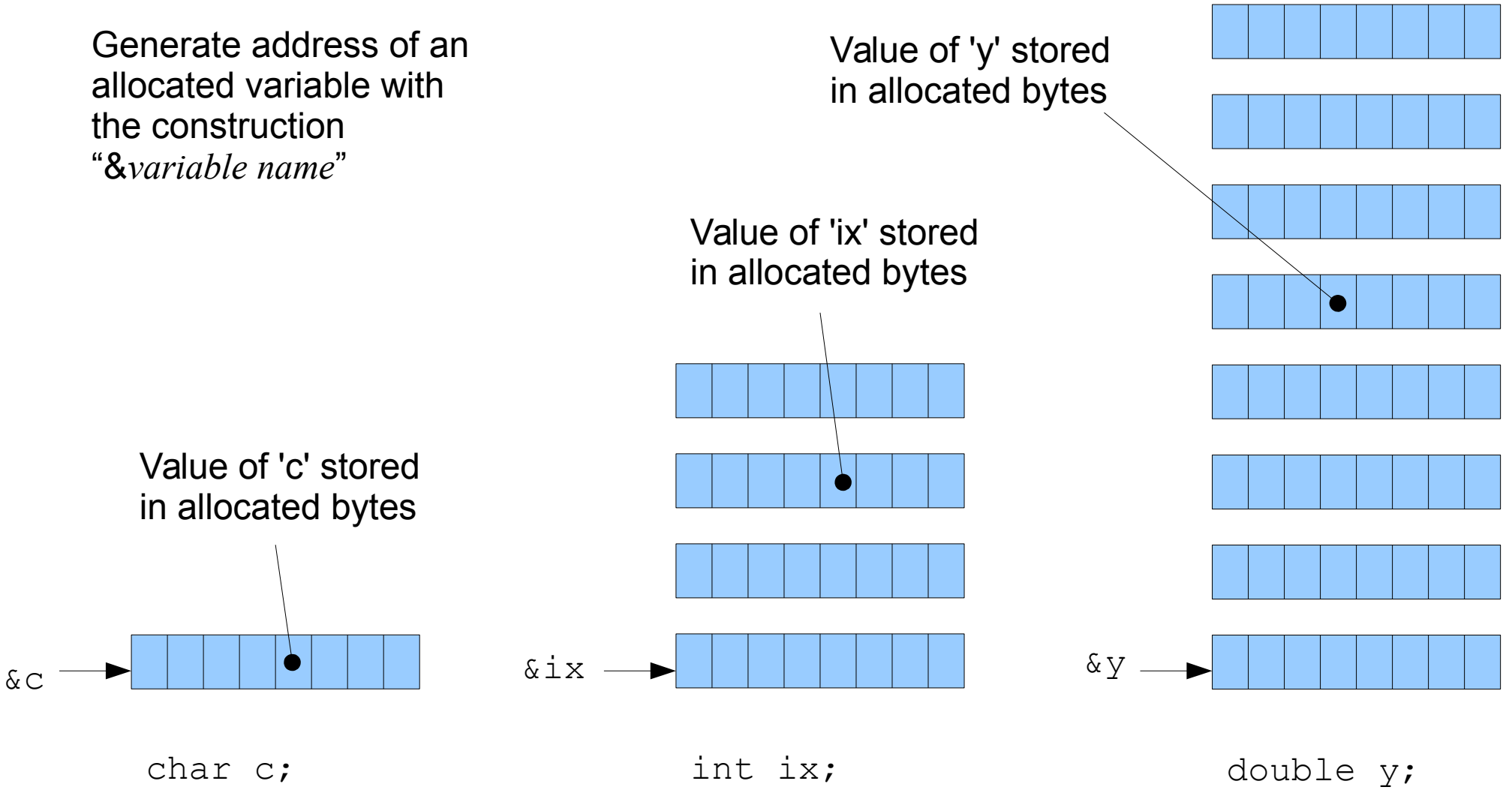
The declaration above means "the variable dereferenced by `ptr` is a char", so `ptr` must be a pointer to char.

Image for C Variables and Pointers

Assigning a pointer variable is like writing down the box number label for one box and stashing it into another box

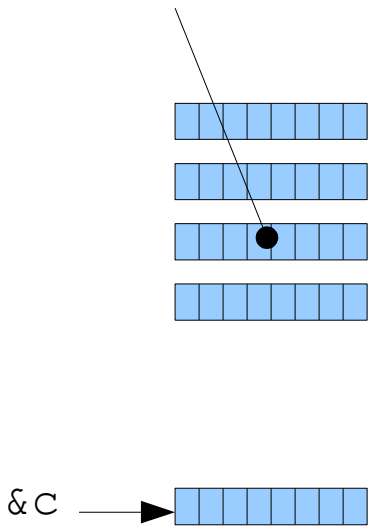
Allocation of C Data Types in RAM

Generate address of an allocated variable with the construction "&*variable name*"



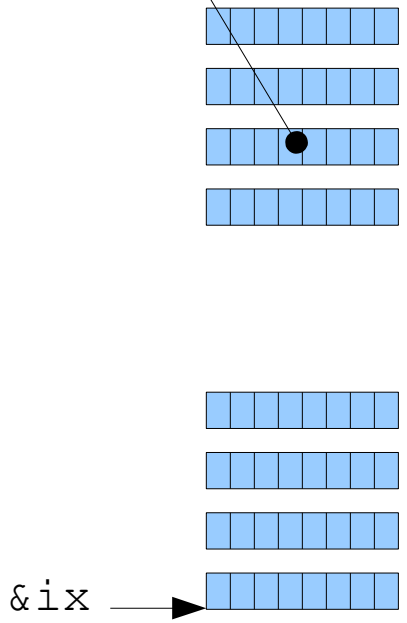
Assignment to Pointer Variables

Value of 'pc', which is the address of 'c', stored in allocated bytes



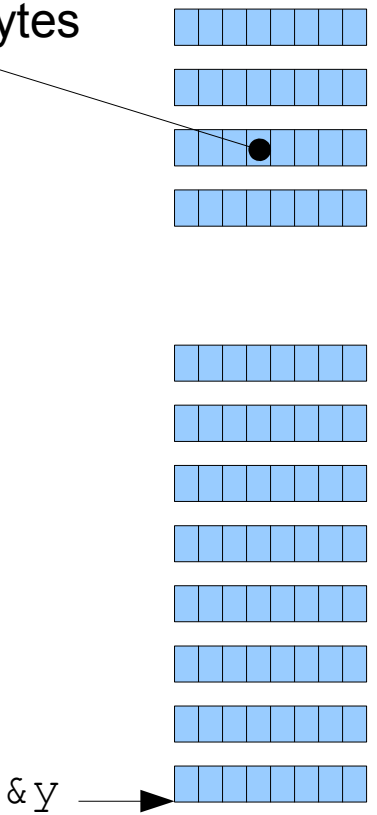

```
char c,*pc;
pc = &c;
```

Value of 'pix', which is the address of 'ix', stored in allocated bytes




```
int ix,*pix;
pix = &ix;
```

Value of 'py', which is the address of 'y', stored in allocated bytes




```
double y,*py;
py = &y;
```

Why Use Pointer Variables?

A single pointer variable can hold the address of many bytes of data in memory, and can be used to efficiently point a function at large blocks of data.

A pointer passed as an argument to a function allows the function to change the value of the variable it points to.

Pointers are used to easily handle memory allocated at run time to accommodate large amounts of data of variable size.

Languages like Fortran and Java make the assumption it knows when you want a pointer and when you want the data itself. That choice is left to you in C.

Allocation of C Arrays in RAM

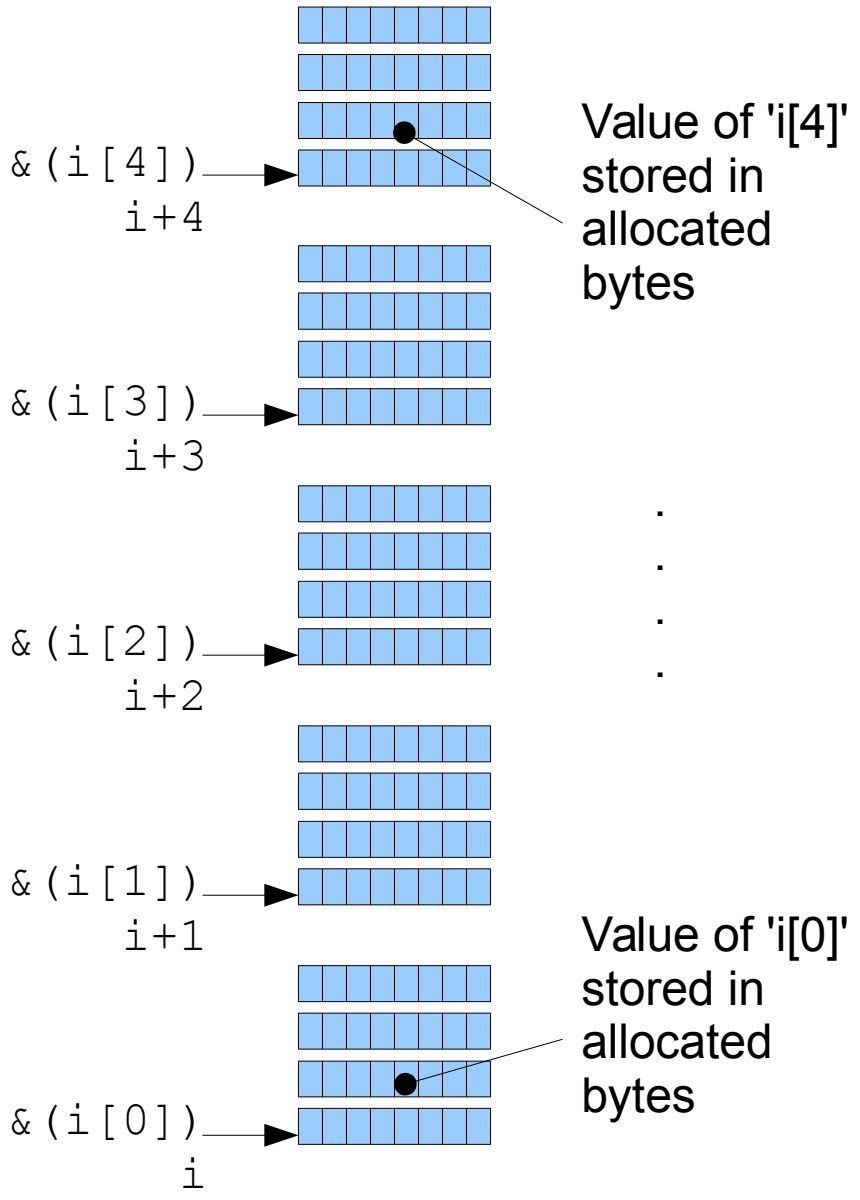
Example: define an array of five integers with

```
int i[5];
```

Individual array elements are referenced as
`i[0], i[1], i[2], i[3], i[4]`

The symbol 'i' is defined to have a value of a pointer to the first element of the array

Note pointer arithmetic adds to the memory address according to the size of the object it points to

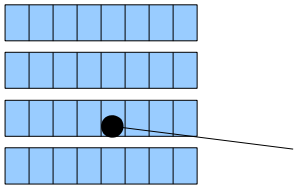


Storing Character Strings as C Arrays

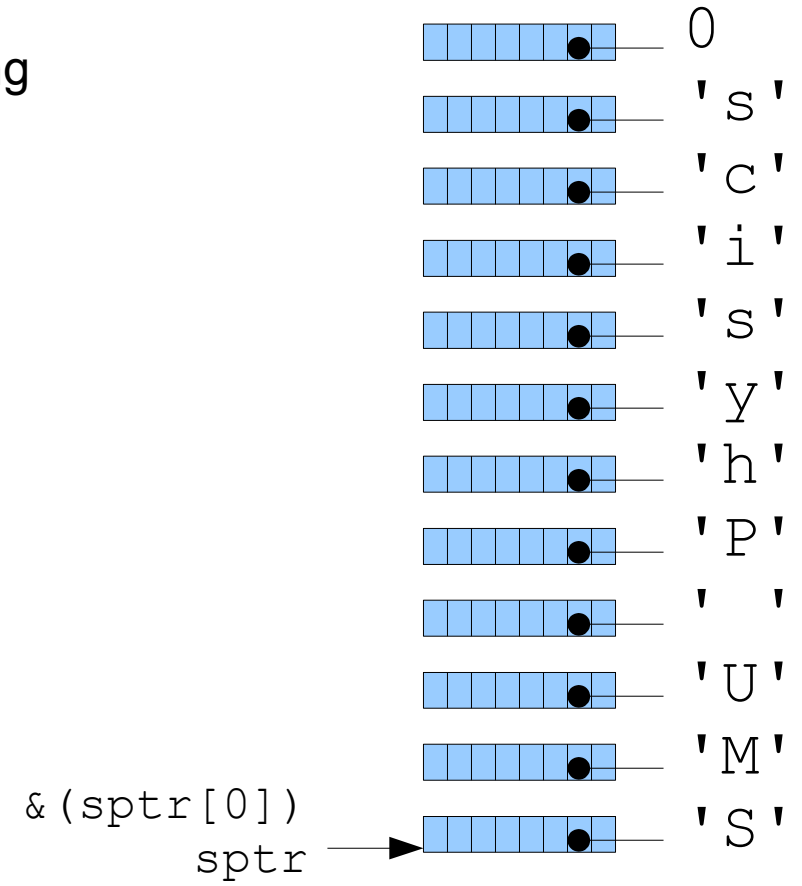
Example: define a pointer to a character string and assign the address of a constant character array to it

```
char *sptr;
sptr = "SMU Physics";
```

Individual string characters are referenced as `sptr[0], sptr[1], sptr[2], ...`



Value of 'sptr' stored in allocated bytes

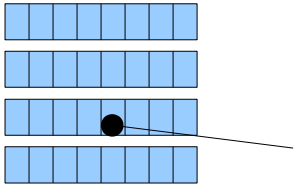


Main Program Command Line Arguments

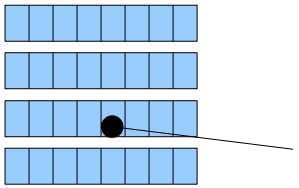
Program “code” is invoked with the command line:

```
code one two three four

int main(int argc, char *argv[]) {
    ...
```

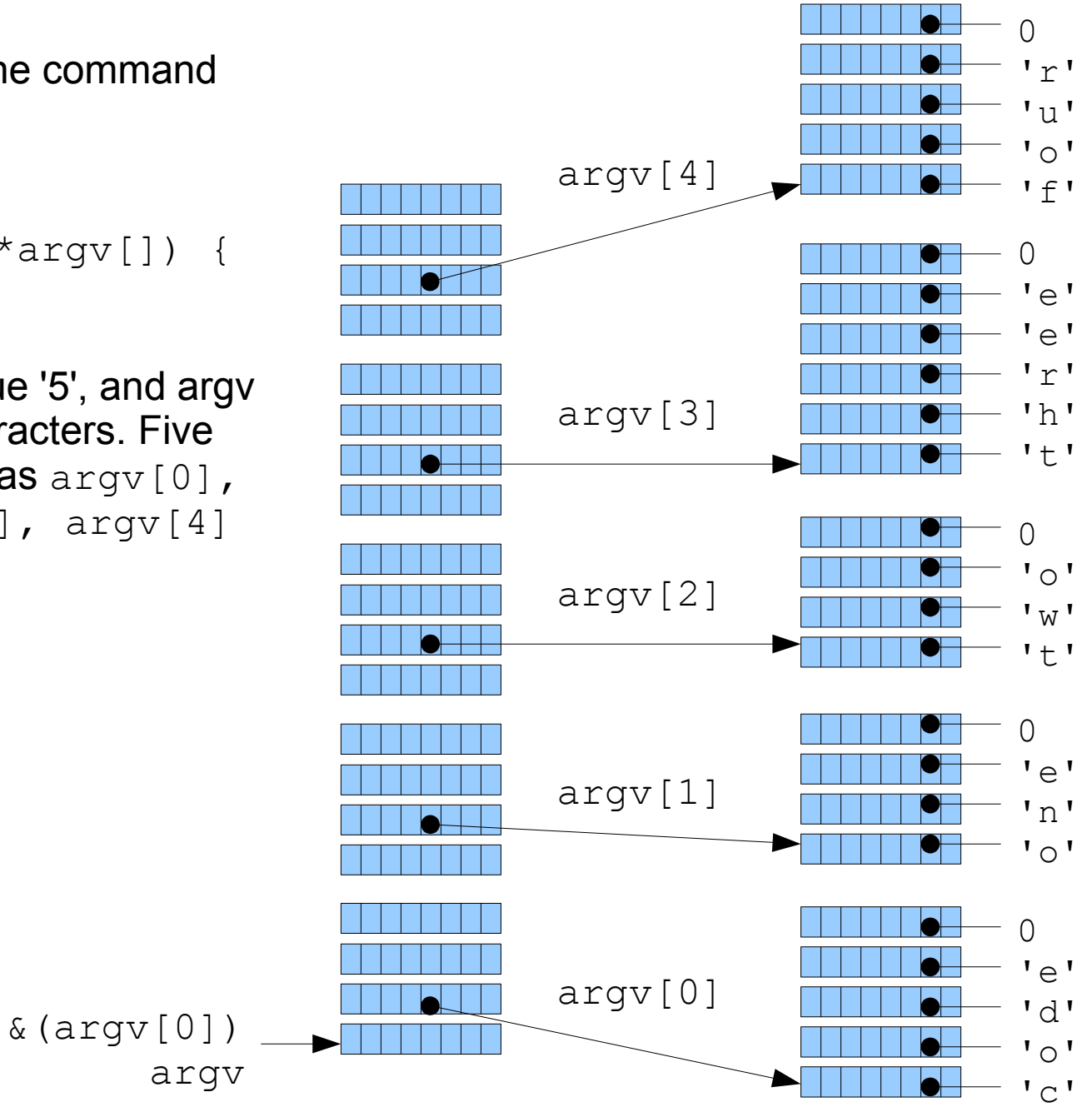
Integer 'argc' is assigned the value '5', and argv is an array of five pointers to characters. Five argument strings are referenced as `argv[0], argv[1], argv[2], argv[3], argv[4]`



Value of 'argv' stored in allocated bytes



Value of 5 stored in allocated bytes

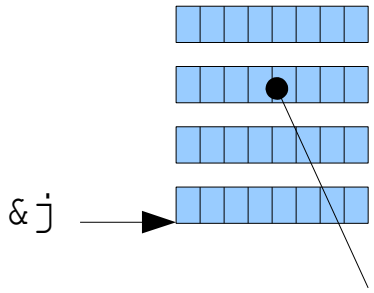


Translating Character Strings into Numerical Variables

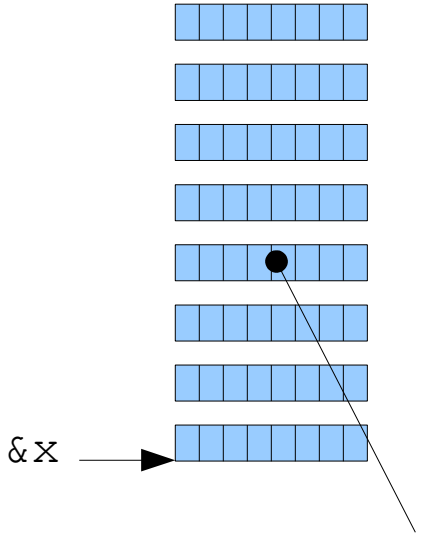
Program "code" is invoked with the command line:

```
code 42 1.23456e7
```

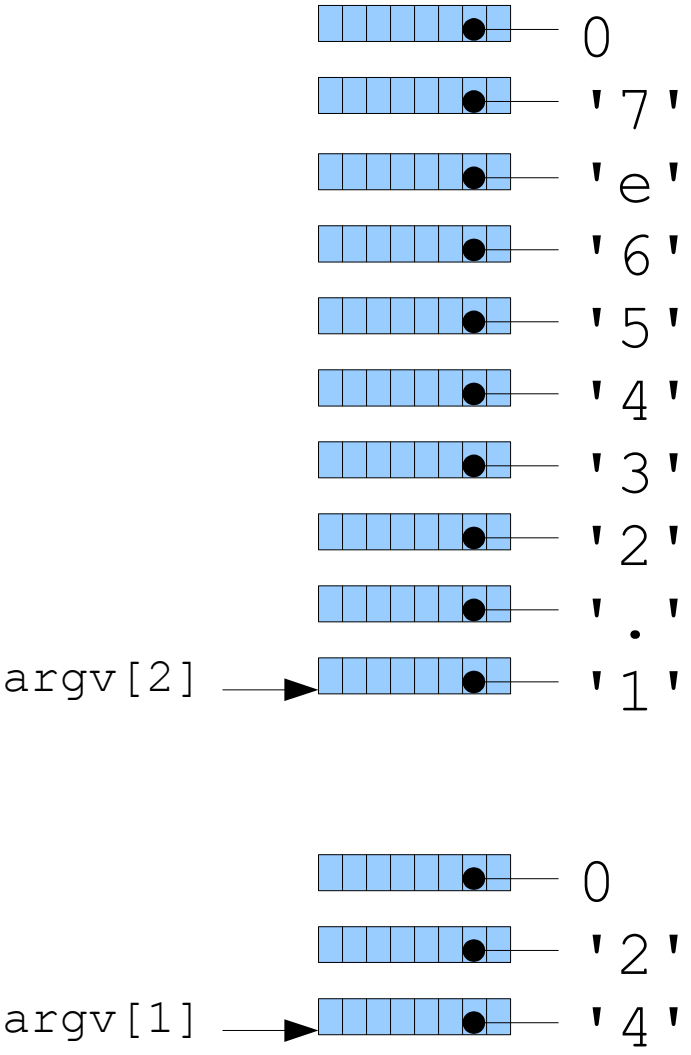
```
int main(int argc, char *argv[]) {
  int j;
  double x;
    ...
  j = atoi(argv[1]);
  x = atof(argv[2]);
```



42 stored in allocated bytes



1.23456×10^7 stored in allocated bytes



Comments in C Code

Any text between the character combination `/*` and `*/` is ignored.

Use this to liberally sprinkle your C code with comment text to improve readability.

Also the C++ style comments are also recognized,
so any text after the character combination `//` to the end of the line is ignored.

Examples:

```
/* Reset all velocities to zero */
for (i=0; i<n; i++) {
  v[i] = 0.0;
}
```

```
r = sqrt(x*x + y*y + z*z);   /*calculate radial distance to charge*/
phi = (0.25 / (PI * permittivity)) * (q / r);  // calculate potential
```

Extreme Examples of Unreadable C Code

The International Obfuscated C Code Contest
http://www.ioccc.org/

```
d b,x,h;o q[9802],f[9802];void w(d i){fflush(stdout);printf(
"%c"  ,   (  i>b)?(i%b>0?q[i-b-1]:10):5*(b==i?2:9));if(i<x+b
-     1    )w(i+1) ; } d p(o*e,d i ) { e +=i;n h = ( - b-1
)       [                e ] %        2 + ( -          b
) [ e ] %    2                                          +
(-b+1)[ e           ]  % 2         +(       -     1  ) [  e
]  %  2+  e  [1   ]% 2 +b [  e   -  1]% 2  +    b[ e ]% 2  +
b    [      e+1 ]     %2      ;}d m (  d      i     ){    n
(  i  == x -       1      )      ?0     :(  f    [      i  +
1  ]=m                (i +       1) ) ,32           +3 /  (
(    p( q  ,   i  )>  3      ||    h <  2       )?  7      :
( ( h                  ==        2 &&q[ i ]          ==  32
     )?    8 :            1 )     );}d                    y
(       d                       i ){ d   j , s , t ,a  ,u
;if (     x - 1== i ) n 1 ;if ( f           [i]==2)       {
f [i ]     <<=4;if(y( i))n 1;f[i           ]-=- 3 ; if     (
y(i )       )n 1;n  0 ; };if((i % b             )     ==
            0 ||(       i % b          )   ==           b-
1         )              n        y (i+1          )     ;j
=    -1 ; l          :          ;    j   =j     + 1 ;
if   ( j    >=    ( i  -      i   /  b +       2
   == b          ? 1 :   i       >     2       * b ?i%   b!=
    1    ?2       :    2          *   1<<     1 :8 ))   goto
c;     u       =   p(  f,   i  ) ; if  (        i          <
x  - 2*b - 1   )  { a =( s       =((  t=i       %   b==
       1)           ?    1      :    i % b          !=     b
   - 2     )   )      &&     i   <            2 * b;u+=    (
a               ? (                              j& 4   )
>>    2 : 0 ) +  (t  ?    ( j & 2 ) / 2 : 0)  +            (
s? (   j&1):0);}else a =t=s=0;if((u&2)+(u&4)!=          2 &&
(q[   i]& 2 ) ==2||u==2&& (f[i]+q[i]) % 2   ==1          ||u
== 3            && ( q [   i]&2)== 0                   )
                      goto            l      ;     b
```

```
[i        +   f          ]     ^=       !          t ?     0
 :   b[ i +       f ]  ^ (         10 + (      j &      2  )
/    2  )     * 3+2;b[ i    + f    +  1    -    b  ]  ^=   !
a  ?0 :          b  [i  +  f    +1 -          b]^   (10+  (
j   &    4)          /            4    )*  3   +2 ;    b [ i
+f     +       1  ]  ^= ! s   ?0     :       b  [i+ f  +   1
]  ^ (10 + (   j  &1)            ) *3+ 2 ;      if  ( y (  i
+      1)           ){ n  1  ;       };          (  a      ?
i  + 1     :     0 )             [ f     ]           =(   t
?                    i+                            b       :0
)       [ f                  ] =       (     s ?          i
   + b +1:0) [ f ] =     2 ;goto l ;       c:n 0 ; }       d
  main(d c,o**v){d i; x=b=0;while((q[x++]=getchar()   )   !=
  EOF ){x-=(q[x-1]==10)? b+=1,1:0;q[x-1]^= (q[x-1]        ==
32)? 0  :  q [   x  -        1 ] ^ 35  ; } ; w (   0      );
for                              (                      ;   ;
)       {         z    ( f             ,         2 ,  x *
     k   (  o       )  )  ;   for   (  i  =    0;i<=x -    1;
i  = i +   b  )q [   i /    b]   = q [   x      -  b+  i  /b
]   =    q       [ i  ]=  q       [    i          -       (
(i   ==  0)          ?           0:  1)       ]  =f [ i  /
b   ] = f[ x      -b -1+ i/  b]   = f [i ]      =f[
      i    -          (             (    i        == 0  )  ?
0       :1        ) ]  =        32;   if       ( c        ==
1           ) m (         0                          );   else
if          (y (     b                       +             1)
==0)n 1;f[0]<<=1<<2;memcpy(q,f,x*k(o));w(0);sleep(1);};n 0;}
```

Program to Display Bits of double Data

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

int main(int argc,char *argv[]) {
  int j;
  double x,*px;
  unsigned char *pc;

  if (argc >= 2) x = atof(argv[1]);
  else {
    fprintf(stderr,"Insufficient command line arguments\n");
    exit(1);
  }
  px = &x;
  printf("x=%lf px=%p sizeof(double)=%d\n",x,px,sizeof(double));
  pc = (unsigned char *) px;
  j = sizeof(double) - 1;
  while (j >= 0) {
    printf("byte %d at %p = 0x%02X\n",j,pc + j,pc[j]);
    j--;
  }
  exit(0);
}
```

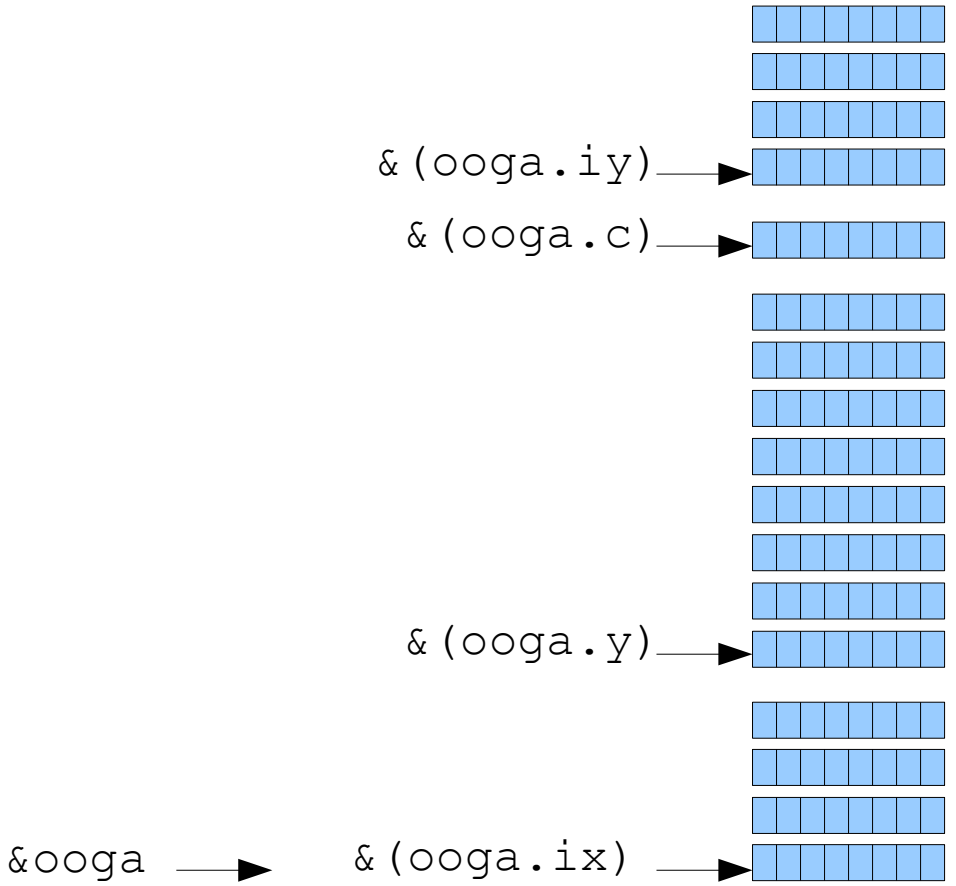
Example Output of double Bits

```
$ ./double_bits 1.0
x=1.000000 px=0xbf9eec08 sizeof(double)=8
byte 7 at 0xbf9eec0f = 0x3F
byte 6 at 0xbf9eec0e = 0xF0
byte 5 at 0xbf9eec0d = 0x00
byte 4 at 0xbf9eec0c = 0x00
byte 3 at 0xbf9eec0b = 0x00
byte 2 at 0xbf9eec0a = 0x00
byte 1 at 0xbf9eec09 = 0x00
byte 0 at 0xbf9eec08 = 0x00
$ ./double_bits 0.1
x=0.100000 px=0xbf8ddf98 sizeof(double)=8
byte 7 at 0xbf8ddf9f = 0x3F
byte 6 at 0xbf8ddf9e = 0xB9
byte 5 at 0xbf8ddf9d = 0x99
byte 4 at 0xbf8ddf9c = 0x99
byte 3 at 0xbf8ddf9b = 0x99
byte 2 at 0xbf8ddf9a = 0x99
byte 1 at 0xbf8ddf99 = 0x99
byte 0 at 0xbf8ddf98 = 0x9A
```

Allocation of C structures in RAM

```
struct {
   int ix;
   double y;
   char c;
   int iy;
} ooga;
```

Individual structure elements are referenced as `ooga.ix, ooga.y, ooga.c, ooga.iy`



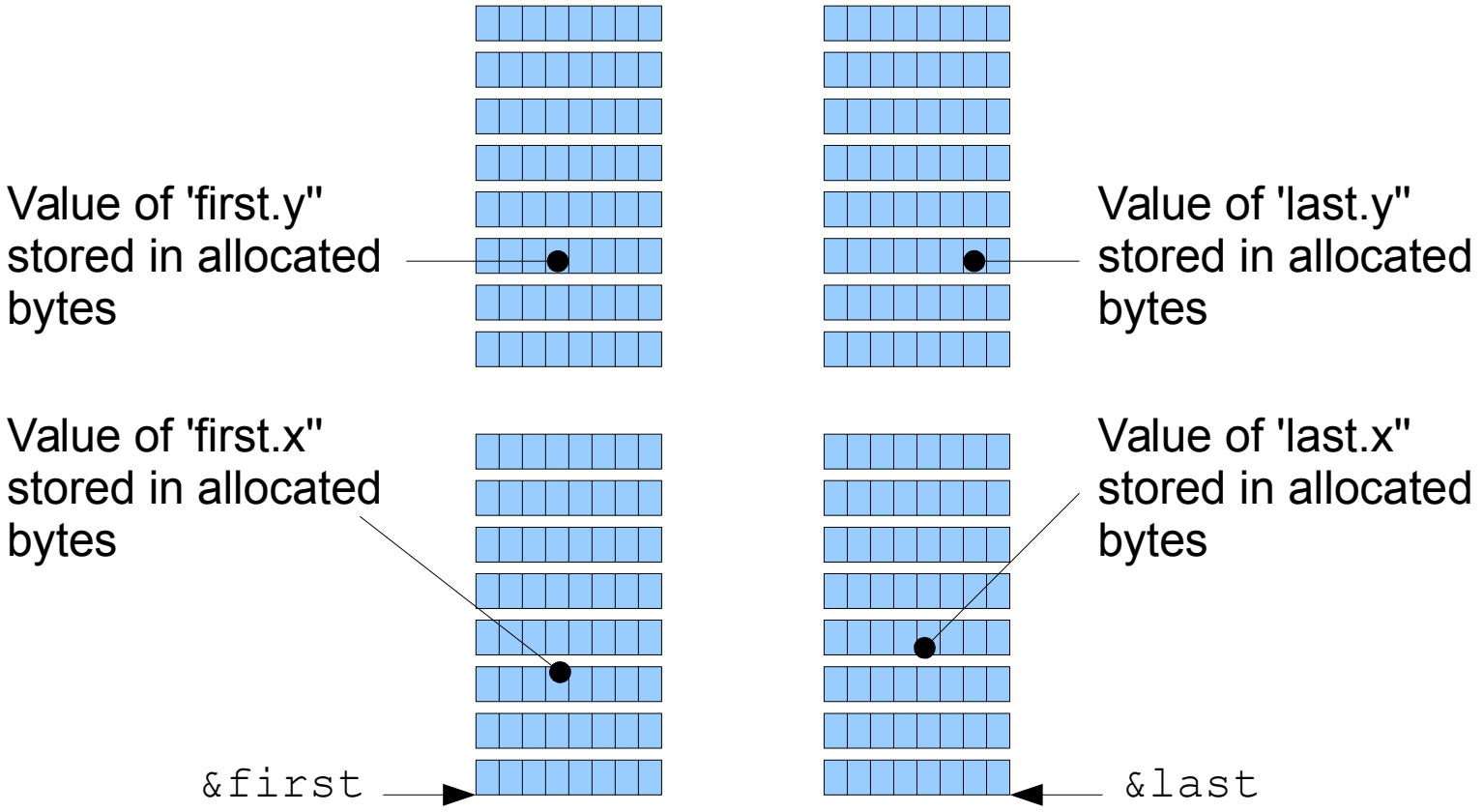
Use Structures to Define New Data Types

Define structure type by including a structure name, but do not allocate memory here

```
struct point2d {
  double x,y;
};
```

Allocate memory here for two instances of the structure

```
struct point2d first,last;
```



Structures Can Have Pointers To Them

Define structure type by including a structure name, but do not allocate memory here

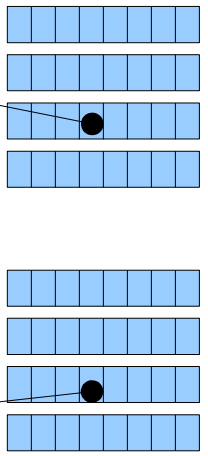
```
struct point2d {
  double x,y;
};
```

Allocate memory here for two instances of the structure and two pointers to the structure

```
struct point2d first,last,*pf,*pl;

pf = &first;
pl = &last;
```

Value of 'pl", the address of 'last', stored in allocated bytes

Value of 'pf', the address of 'first" stored in allocated bytes



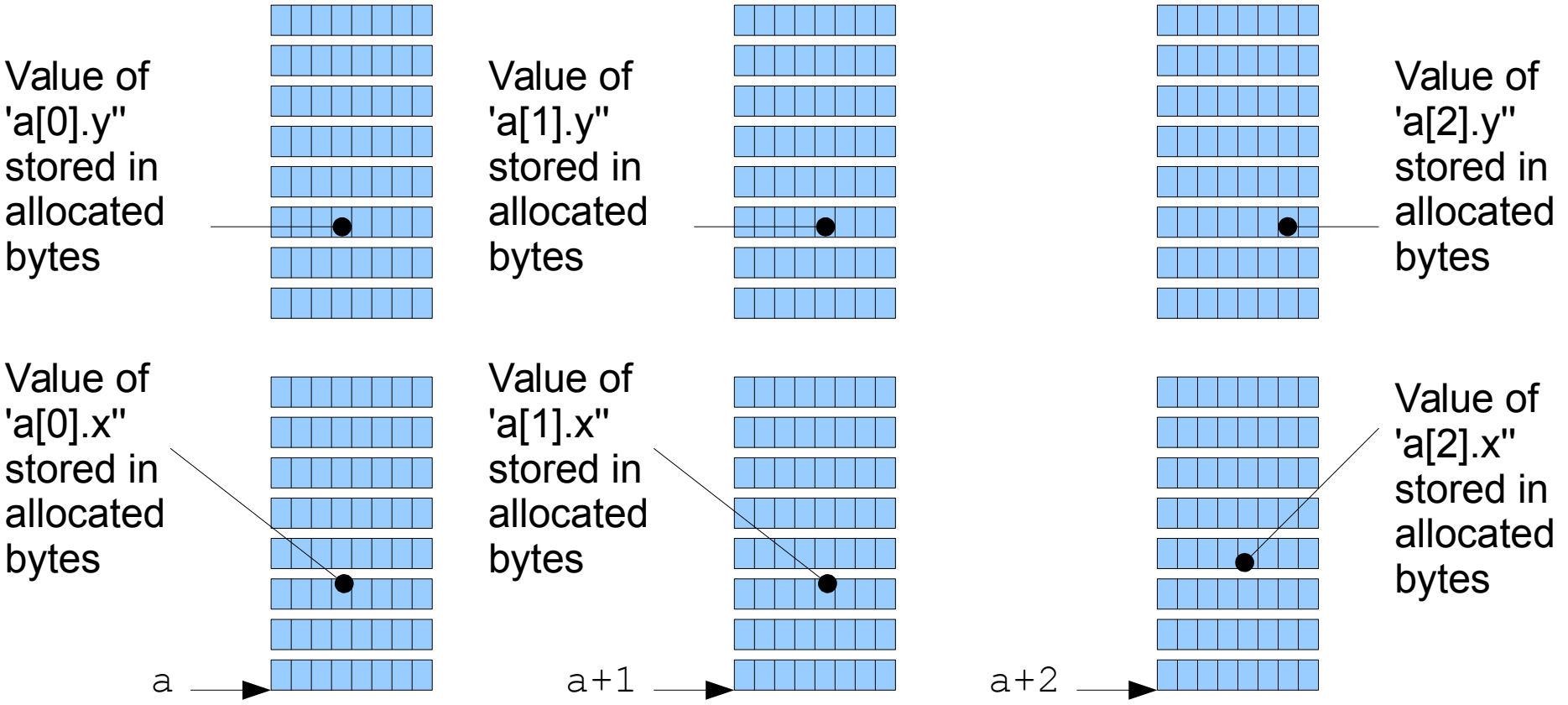
Individual structure elements can referenced as `(*pf).x`, `(*pf).y`, `(*pl).x`, `*(pf).y` or as `pf->x`, `pf->y`, `pl->x`, `pl->y`

Arrays of Structures

Define structure type by including a structure name, but do not allocate memory here

```
struct point2d {
  double x,y;
};

struct point2d a[3];
```

Allocate memory here for a three-element array of the structure



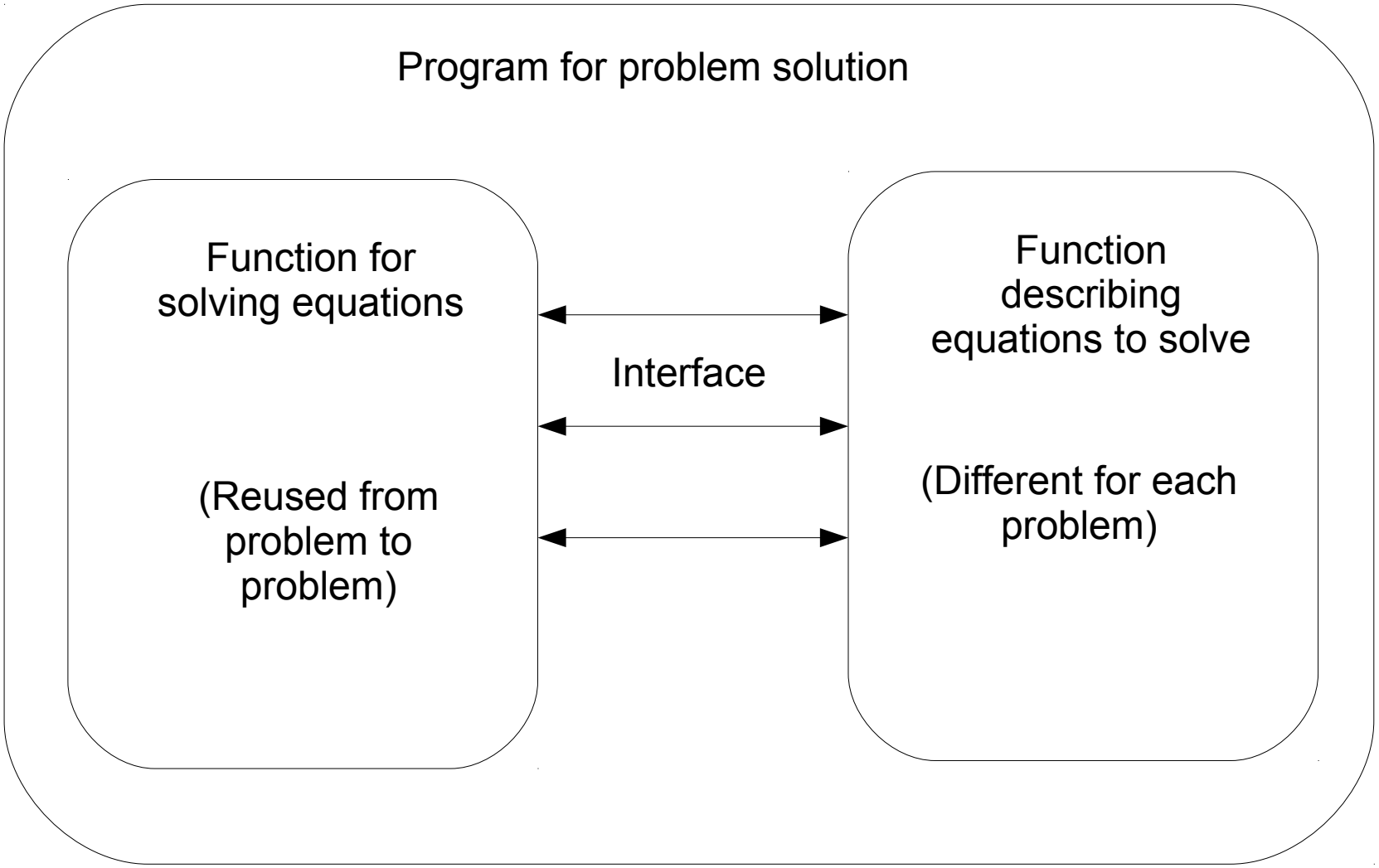
Program 3 to Plot Damped Oscillator Response

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "constants.h"

int main(int argc,char *argv[]) {
  double t,disp,f0,w0,Q,tmax,tstep;

  if (argc != 4) {
    fprintf(stderr,"%s <f0> <Q> <tmax>\n",argv[0]);
    exit(1);
  }
  f0 = atof(argv[1]);
  w0 = TWO_PI * f0;
  Q = atof(argv[2]);
  tmax = atof(argv[3]);
  tstep = tmax * 0.01;
  tmax = tmax + (tstep * 0.5);
  t = 0.0;
  while (t < tmax) {
    disp = exp(-0.5 * w0 * t / Q) * cos(w0 * t);
    printf("%.8g %.8g\n",t,disp);
    t = t + tstep;
  }
  exit(0);
}
```

Include File with Physical Constants

```
#define BOLTZMANN 1.380648813e-23
#define PLANK 6.6260695729e-34
#define PLANK_REDUCED 1.05457172647e-34
#define GRAVITY 6.67428e-11
#define GRAVITATIONAL_ACCEL 9.80665
#define CHARGE 1.602e-19
#define PERMITTIVITY 8.854e-12
#define PI 3.14159265358979323846264338327950288
#define TWO_PI 6.283185307179586
#define FOUR_PI 12.56637061435917
#define HALF_PI 1.570796326794897
#define QUARTER_PI 0.7853981633974483
#define RADTODEG 57.29577951308234
#define DEGTORAD 0.0174532925199433 29
```

Reusing Program Code



Program 4 to Plot Damped Oscillator Response

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "constants.h"
#include "sweep.h"

double w0,Q;

double osc(double t) {
  double disp;
  disp = exp(-0.5 * w0 * t / Q) * cos(w0 * t);
  return(disp);
}

int main(int argc,char *argv[]) {
  double f0,tmax,tstep;

  if (argc != 5) {
    fprintf(stderr,"%s <f0> <Q> <tmax> <tstep>\n",argv[0]);
    exit(1);
  }
  f0 = atof(argv[1]);
  w0 = TWO_PI * f0;
  Q = atof(argv[2]);
  tmax = atof(argv[3]);
  tstep = atof(argv[4]);
  sweep(osc,0.0,tmax,tstep);
  exit(0);
}
```

Argument Sweeping Function and Include File

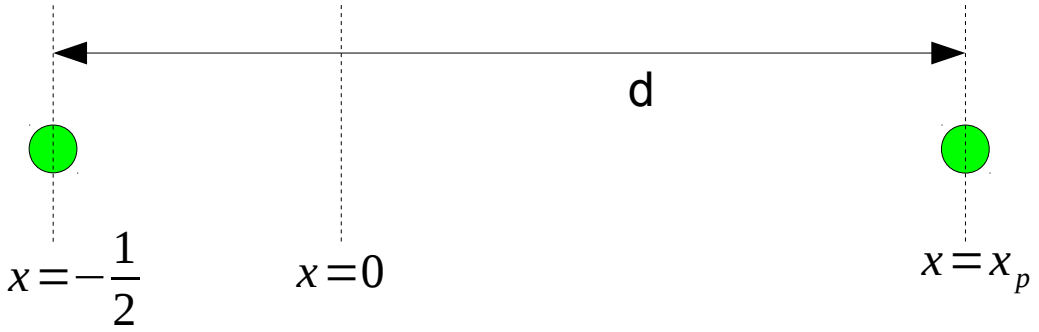
```
#include <stdio.h>
#include <stdlib.h>
#include "sweep.h"

int sweep(double (*func)(double x),double xstart,double xstop,double xinc) {
  double x;

  xstop = xstop + (xinc * 0.5);
  x = xstart;
  while (((xinc > 0.0) && (x < xstop)) || ((xinc < 0.0) && (x > xstop))) {
    printf("%.8g %.8g\n",x,(*func)(x));
    x = x + xinc;
  }
  return(0);
}
```

```
int sweep(double (*func)(double x),double xstart,double xstop,double xinc);
```

Program to Plot Potential from Two Charges

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

int main(int argc,char *argv[]) {
  double x,y,z,d1,d2;

  x = 0.0;
  while (x < 10.001) {
    y = 0.0;
    while (y < 10.001) {
      d1 = sqrt(0.05 + ((x-7.5)*(x-7.5)) + ((y-2.5)*(y-2.5)));
      d2 = sqrt(0.05 + ((x-2.5)*(x-2.5)) + ((y-7.5)*(y-7.5)));
      z = (10.0 / d1) - (5.0 / d2);
      printf("%.8g %.8g %.8g\n",x,y,z);
      y = y + 0.1;
    }
    putchar('\n');
    x = x + 0.1;
  }
  exit(0);
}
```

From System to Pseudocode to C Code



Pseudocode:

$$\text{real}\, x, x_p, d$$
$$x_p \leftarrow \cdots$$
$$\vdots$$
$$d \leftarrow x_p + \frac{1}{2}$$

C code:

```
double x,xp,d;

d = xp + (1/2);        <-- Bug!
d = xp + (1.0/2.0);    <-- Better
d = xp + 0.5;          <-- Best
```

Integer Constants in Floating Point Expressions

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

static double x,y;

int main(int argc,char *argv[]) {
  x = 5.0;
  y = x + 1/2;
  printf("x=%.2f y=%.2f\n",x,y);
  y = x + 0.0;
  printf("x=%.2f y=%.2f\n",x,y);
  y = x + 1.0/2.0;
  printf("x=%.2f y=%.2f\n",x,y);
  y = x + 0.5;
  printf("x=%.2f y=%.2f\n",x,y);
  exit(0);
}
```

Output:

```
x=5.00 y=5.00
x=5.00 y=5.00
x=5.00 y=5.50
x=5.00 y=5.50
```

Integer Constants in Floating Point Expressions

Look at compiled binary program with objdump tool:

```
...
  x = 5.0;
  11:   dd 05 10 00 00 00       fld    QWORD PTR ds:0x10
  17:   dd 1d 00 00 00 00       fstp   QWORD PTR ds:0x0
  y = x + 1/2;
  1d:   dd 05 00 00 00 00       fld    QWORD PTR ds:0x0
  23:   d9 ee                   fldz
  25:   de c1                   faddp  st(1),st
  27:   dd 1d 08 00 00 00       fstp   QWORD PTR ds:0x8
...
  y = x + 0.0;
  4f:   dd 05 00 00 00 00       fld    QWORD PTR ds:0x0
  55:   d9 ee                   fldz
  57:   de c1                   faddp  st(1),st
  59:   dd 1d 08 00 00 00       fstp   QWORD PTR ds:0x8
...
  y = x + 1.0/2.0;
  81:   dd 05 00 00 00 00       fld    QWORD PTR ds:0x0
  87:   dd 05 18 00 00 00       fld    QWORD PTR ds:0x18
  8d:   de c1                   faddp  st(1),st
  8f:   dd 1d 08 00 00 00       fstp   QWORD PTR ds:0x8
...
  y = x + 0.5;
  b7:   dd 05 00 00 00 00       fld    QWORD PTR ds:0x0
  bd:   dd 05 18 00 00 00       fld    QWORD PTR ds:0x18
  c3:   de c1                   faddp  st(1),st
  c5:   dd 1d 08 00 00 00       fstp   QWORD PTR ds:0x8
```

5.0 stored at hex 10 = 16

x stored at hex 00 = 0

Floating point load 0! Bug!

y stored at hex 08 = 8

Load floating point 0

0.5 stored at hex 18 = 24

fld=floating point load stack
fldz=load stack with zero
faddp=add and pop stack
fstp=store and pop stack

Actual Lines of Code from Homework

Probably will get away with it, but poor form!

```
double a,b,x;
x = x*b*(1+a)/(1+b);
```

The slippery slope downhill!!

```
double a,n = 1;
e_approx = pow((1 + 1/n),n);
a = 0.5*(t + 1/t);
```

```
x = 2 + pow(2,1/2);
```

Potential disaster!!!

```
y = 1/M_PI*cos(1.0*x-1.0*sin(x));
```

Totally trashes the computation!!!

Class Rule for Arithmetic Expressions

- Decide if an arithmetic assignment statement is to produce an integer or floating point result
- If floating point, all constants in the expressions must be coded as floating point, that is, including a decimal point or exponent

Do You Really Need to Call the `pow()` Function?

Calling `pow(x,y)` is equivalent to `exp(y * log(x))` !!!
This is lots of computation that is rarely actually needed. Most beginning numerical programmers overuse this expensive function, instead of much faster expressions such as

```
x*x

x*x*x

sqrt(x)
```

etc....