C programming I

Karl W Broman
Department of Biostatistics
Johns Hopkins University

http://www.biostat.jhsph.edu/~kbroman

Acknowledgment: I borrowed a great deal from Phil Spector, Univ. California, Berkeley

Why program in C?

Speed!

A poorly written C program can be faster than the most efficient R or Perl program.
C, Perl, and R

<table>
<thead>
<tr>
<th>Language</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Serious computation</td>
</tr>
<tr>
<td>Perl</td>
<td>Text manipulation</td>
</tr>
<tr>
<td>R</td>
<td>Graphics and interactive analysis</td>
</tr>
</tbody>
</table>

A serious statistician should be fluent in all of these languages!

How to learn a programming language

- Get a good book.
- Experiment on the computer.
- Find some good problems to work on.
- Look at others’ code.
- Use the language routinely.
Suggested books

- Reek (1997) *Pointers on C*. Addison-Wesley.

A first, simple program

```c
/* proglA.c */
main()
{
    double x, y;

    printf("Enter first number: ");
    scanf("%lf", &x);

    printf("Enter second number: ");
    scanf("%lf", &y);

    printf("The sum is %f\n", x+y);
}
```
Actually should be...

```c
#include <stdio.h> /* contains function declarations */

int main() /* returns an integer */
{
    double x, y;

    printf("Enter first number: ");
    scanf("%lf", &x);

    printf("Enter second number: ");
    scanf("%lf", &y);

    printf("The sum is %f\n", x+y);

    return(0); /* exit normally (no errors) */
}
```

---

**Compiling the program**

Say the program is in the file `myprog.c`.

**You compile** it by typing

```
gcc -o myprog myprog.c
```

or typing

```
gcc -Wall -o myprog myprog.c -lm
```

This produces an executable, `myprog`, which you can execute just typing its name.

**Note:** `-Wall` results in extra warning messages; `-lm` indicates to link the math library (for stuff like `sqrt` and `sin`).

The program `lint` (see also `splint`) analyzes C programs for potential problems.
Declarations

**char** — character variable (a one-byte integer)
**int** — “natural” integer
**long** — the longest integer available
**short** — the smallest (potentially) integer

Also: **unsigned int**, **unsigned long**, **unsigned short**

**float** — (potentially) single precision
**double** — largest floating point number available

**void** — for functions which don’t return a value

All functions should be declared!

For example:

```c
double myfunc(double x, double y);
long secfunc(void);
```

Arrays

Any type of object may be declared as an array.

```c
double x[10];
```

The elements: `x[0], x[1], ..., x[9]`

The array index is an offset in memory.
Header files

Pull in function declarations. (No executable code!)

Found in /usr/include
  (common subdirectory /usr/include/sys)

For example:
  math.h — common math functions
  stdio.h — print functions
  stdlib.h — general utilities

Put (at the top of the program)
  #include <math.h>

Put declarations for your own functions in your own header file
(e.g., myprog.h) and use the following:
  #include "myprog.h"

Constants

integer: Just use the number (e.g., 21)

long integer: Use an “L” after the number (e.g., 21L)

double: Use the number with a decimal point or E-notation
  (e.g. 21. or 21.0 or 21e0)

float: Use the number and “f” or “F”. (e.g. 21f)

character: Use single quotes (e.g. ‘a’)

  special characters: ‘\n’    ‘\t’    ‘\’    ‘\‘    ‘\’

character strings: Use double quotes (e.g., "a string")

  Strings are arrays of characters terminated by ‘\0’
  – ‘\0’ is the character whose value is 0
  – Note that ‘\0’ \≠ ‘0’
Operators

- **Arithmetic operators**
  
  Binary:  
  
  +  −  *  /  %

  Urinary:  −  (change sign)

  **Note**: `pow` does exponentiation:
  
  ```c
  double pow(double, double) (see <math.h>)
  ```

  e.g., `x^3 = pow(x, 3.0);`

- **Relational operators**

  ```c
  >  >=  <  <=  ==  !=
  ```

  e.g., `j = (x>3.0);`  
  (Be careful about `=` versus `==`)

- **Logical operators**

  ```c
  !  &&  ||
  ```

**Combining logical operators**

```c
if(i > 10 && (x=getthat()) ) { ... }
```

**Note**: if `i>10` is FALSE, `getthat()` won’t be called.

“Side effect”: e.g., in an `&&` statement, the thing on the right is only evaluated if the thing on the left is TRUE.

**Suggestion**: Use parentheses freely.

**Suggestion**: In using `==`, `<=`, etc., try to ensure that both sides are of the same variable type.
Type casting

```c
#include <math.h> /* prog2.c */

int i;
double x;

i=3;

x = sqrt((double)i);
/* NOT: x = sqrt(i); */
```

**Suggestion:** If you are in doubt about how a computer will interpret a number, go ahead and use the casts; it won’t slow down your program.

Increment, decrement and assignment

```
i = i + 1;    i += 1;    i++;
i = i - 1;    i -= 1;    i--;
```

**Postfix version** (e.g., i++): use the variable’s value, then increment or decrement.

**Prefix version** (e.g., ++i): Increment or decrement, then use the variable’s (new) value.

```
if(++nobs > 10) { ... } versus
if(nobs++ > 10) { ... }
```

**Suggestion:** Avoid constructions where the prefix vs. postfix version matters.
Bitwise operators

&    bitwise and
|    bitwise or
^    bitwise exclusive or
<<   left bitshift
>>   right bitshift
~    unary ones complement (1 → 0 and 0 → 1)

If-else statements

/*1*/ if(x > 0) x *= -1.0;

/*2*/ if(x > 0) {
    x = sqrt(x);
    y /= x;
}

/*3*/ if(n>0) {
    for(i=0; i<n; i++)
        if(fabs(s[i]) > 1e-8)
            y[i] = z[i]/s[i];
} else printf("n is 0\n");

Suggestion: Use { } freely.
**while loop**

/*1*/ while(x > y) {
    ...
}

1. Evaluate expression
2. If TRUE, execute statements in the block and go back to top
3. If FALSE, skip on to the next thing.

/*2*/ while(1) { ... } /* infinite loop */

/*3*/ while(x>y); { /* an error! */
    ...
}

**do-while loop**

do {
    ...
} while(x > y);

1. Execute statements.
2. Evaluate statements in the block.
3. If TRUE, go back to the top.
4. If FALSE, skip on to the next thing.
for loop

expr1;
while(expr2) {
    ...
    expr3;
}

↕

for(expr1; expr2; expr3) {
    ...
}

1. Execute expression `expr1`.
2. Evaluate expression `expr2`.
3. If TRUE, execute the block, execute expression `expr3`, and go back to 2.
4. If FALSE, skip on to the next thing.

Example for loops

/*1*/     sum = 0.0;
for(i=0; i<n; i++) sum += x[i];

/*2*/     for(i=0; i<n && sum < 100; i++)
    sum += x[i];

/*3*/     for(i=0, j=0; i<n && j<n; i++, j++) { ... }

/*4*/     for( ; i<n; i++) { ... }

/*5*/     r = unif_rand();
for(i=0; i<n; i++) {
    if(r < p[i]) return(i+1);
    else r -= p[i];
}     
return(n); /* this shouldn’t happen */
break and continue

**break**: Immediately exit from a loop.

**continue**: Jump to the next iteration of a loop

```c
i=0; /* prog3.c */
while(1) {
    i++;
    if(i % 2) continue;
    printf(" %2d\n", i);
    if(i>10) break;
}
```

---

**Pointers**

A **pointer** is a variable whose value is the **address** of an object in memory.

Every object type (e.g., `int`, `double`) has its own separate type of pointer.

```c
double *x; /* pointer to double */
int   *y;  /* pointer to integer */

double *z, w; /* z is pointer to double */
/* w is a double */
```
Pointer operators

&  Address operator
When applied to an object, returns the address of that object

*  Dereference operator
When applied to a pointer to an object, returns the value at the address “pointed to.”

double x;  /* prog4.c */
double *dp;

x = 12.0;
dp = &x;

printf("%g\n", *dp);
printf("%x\n", dp);  /* should put (unsigned int)dp */

Uses of pointers

1. Passing arguments that need to be modified.
   In C, the values of arguments are passed to functions. Thus, a function usually only has access to copies of its arguments. If you send pointers as arguments, you can go to the addresses and change the values there.

   void swap(int *a, int *b)  /* prog5.c */
   {
      int temp;

      temp = *a;
      *a = *b;
      *b = temp;
   }

   Call this function as follows:
   int i=3, j=5;
   swap(&i, &j);
2. **Pointers vs. arrays**

If we declare an array

```c
int ix[10];
```

a location in memory will be found that has enough memory assigned to it to hold 10 integers. `ix` is a pointer to that address in memory. **Subscripting is just a dereference operation.**

*ix* is the same as `ix[0]`.

*(ix+3)* is the same as `ix[3]`.

*(ix+i)* is the same as `ix[i]`.

```c
int ix[10], *ip;

ip=ix;       /* the name of an array is a pointer to */
             /* its first element */

ip += 3;     /* now ip points to the addr of ix[3] */
             /* Note: We can modify ip but not ix. */
```

---

3. **Dynamic memory allocation**

Dynamic memory allocation allows you to find the memory you need, and to associate it with a pointer, while the program is running. So there’s no need for “can’t have more than 10,000 observations.”

**malloc** and **calloc** — allocate memory

**free** — free allocated memory

```c
void *malloc(size_t size);
void *calloc(size_t number, size_t size);
void free(void *ptr);
```
3. **Dynamic memory allocation**: Example

```c
#include <stdlib.h> /* prog6.c */

double *x, *y;

int n;

n=20;

x = (double *)malloc(n*sizeof(double));
y = (double *)calloc(n, sizeof(double));
if(x==NULL || y==NULL) {
    printf("Error: cannot allocate space for x and y\n");
    exit(EXIT_FAILURE);
}

free(x);
free(y);

/* Note: The values of x and y are unchanged */
```

4. **2-dimensional (and multi-dimensional) arrays**

   a. **Use the built-in version.**

```c
#include <stdio.h>

int i, j;
int x[3][6]; /* 3 rows and 6 columns */ /* stored by rows */

for(i=0; i<3; i++) {
    for(j=0; j<6; j++) {
        x[i][j] = i+j;
        printf(" row=%d col=%d addr=%u value=%d\n", i, j, (unsigned int)&x[i][j], x[i][j]);
    }
}
```

**Advantage:** Easy.

**Disadvantages:** Pre-determined size and must be a rectangle.
Uses of pointers

4. 2-dimensional (and multi-dimensional) arrays
   a. Use the built-in version.

   **In your mind**
   
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
</tbody>
</table>

   When you declare an array as x[3][6],
   x[i][j] is interpreted as x[i*6 + j]
   which is the same as *(x + i*6 + j)

   **In memory**
   
   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |10 |11 |12 |13 |14 |15 |16 |17 |

4. 2-dimensional (and multi-dimensional) arrays
   b. Use a one-dimensional array.

   /* prog7b.c */
   
   #include <stdlib.h>
   #include <stdio.h>
   int i, j, nr=3, nc=6, *x;

   x = (int *)calloc(nr*nc, sizeof(int));
   if(x==NULL) { /* error */ }

   for(i=0; i<nr; i++) {
      for(j=0; j<nc; j++) {
         x[i + j*nr] = i+j; /* stored by columns */
         printf(" row=%d col=%d addr=%u value=%d\n", i, j, (unsigned int)(x+i+j*nr), x[i+j*nr]);
      }
   }
   free(x);

   **Advantages:** Dynamic memory allocation, flexible shape.
   **Disadvantages:** Cumbersome, ugly, prone to errors.
4. **2-dimensional (and multi-dimensional) arrays**

**b. Use a one-dimensional array.**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>12</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>7</td>
<td>10</td>
<td>13</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>8</td>
<td>11</td>
<td>14</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In your mind

$x[i + j*3]$ is the same as *(x + i + j*3)

In memory

```c
/* prog7c.c */
#include <stdlib.h>
#include <stdio.h>
int i, j, nr=3, nc=6, **x, *temp;

/* allocate space */
temp = (int *)calloc(nr*nc, sizeof(int));
x = (int **)calloc(nr, sizeof(int *));

if(temp==NULL || x==NULL) { /* error */ }

/* make x point to the beginning of each row */
for(i=0; i<nr; i++) x[i] = temp+i*nc;

for(i=0; i<nr; i++)
    for(j=0; j<nc; j++)
        x[i][j] = i+j;

free(x);
free(temp);
```
Uses of pointers

4. 2-dimensional (and multi-dimensional) arrays
   c. Allocate and parse a block: Another version.

   In your mind

   \[
   \begin{array}{cccccc}
   0 & 1 & 2 & 3 & 4 & 5 \\
   6 & 7 & 8 & 9 & 10 & 11 \\
   12 & 13 & 14 & 15 & 16 & 17 \\
   \end{array}
   \]

   \[x[i][j] \text{ is the same as } *( *( x+i ) +j )\]

   In memory

   \[
   \begin{array}{cccccccccc}
   0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 \\
   \end{array}
   \]

   #include <stdlib.h>
   #include <stdio.h>
   int i, j, nr=3, nc=6, **x;

   /* allocate space for the pointers to each row */
   x = (int **)calloc(nr, sizeof(int *));
   if(x==NULL) { /* error */ }

   /* allocate space for the actual matrix */
   x[0] = (int *)calloc(nr*nc, sizeof(int));
   if(x[0]==NULL) { /* error */ }

   /* make x point to the beginning of each row */
   for(i=1; i<nr; i++) x[i] = x[0]+i*nc;

   for(i=0; i<nr; i++)
       for(j=0; j<nc; j++)
           x[i][j] = i+j;

   free(x[0]); /* have to do this first */
   free(x);
Uses of pointers

4. 2-dimensional (and multi-dimensional) arrays

c. Allocate and parse a block.

Advantages: Dynamic memory allocation, flexible shape, referred to in natural way.

Disadvantages: Stored by rows, a bit of work to set up, a tiny bit of extra memory required.

```c
/* prog7d.c */
#include <stdio.h>
#include <stdlib.h>
int i, j, nr=3, nc=6, **x;

/* allocate space for the pointers to each row */
x = (int **)calloc(nr, sizeof(int *));
if(x==NULL) { /* error */ }

/* allocate space for each row */
for(i=0; i<nr; i++) {
    x[i] = (int *)calloc(nc, sizeof(int));
    if(x[i]==NULL) { /* error */ }
}

for(i=0; i<nr; i++)
    for(j=0; j<nc; j++)
        x[i][j] = i+j;

for(i=0; i<nr; i++) free(x[i]);
free(x);
```
Uses of pointers

4. **2-dimensional (and multi-dimensional) arrays**
   d. Allocate each row or column.

   **In your mind**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
</tbody>
</table>

   x[i][j] is the same as *( x+i ) +j

   **In memory**

   x → x[0]x[1]x[2]

   0  1  2  3  4  5  6  7  8  9  10 11  12 13 14 15 16 17

---

**Uses of pointers**

4. **2-dimensional (and multi-dimensional) arrays**
   d. Allocate each row or column.

   **Advantages:** Dynamic memory allocation, flexible shape, even easier to have columns of different lengths, referred to in natural way.

   **Disadvantages:** Stored by rows, not contiguous memory, a bit of work to set up and take down, a tiny bit of extra memory required.
Example: multi-dimensional array

```c
#include <stdio.h> /* prog8.c */
#include <stdlib.h>
int i, j, k, dim1=4, dim2=5, dim3=3;
int ***x, *temp;

temp = (int *)calloc(dim1*dim2*dim3, sizeof(int));
if(temp==NULL) { /* error */ }

x = (int ***)calloc(dim1, sizeof(int **));
if(x==NULL) { /* error */ }

for(i=0; i<dim1; i++) {
    x[i] = (int **)calloc(dim2, sizeof(int *));
    if(x[i]==NULL) { /* error */ }

    /* make things point properly */
    for(j=0; j<dim2; j++) x[i][j] = temp+(i*dim2+j)*dim3;
}

for(i=0; i<dim1; i++)
    for(j=0; j<dim2; j++)
        for(k=0; k<dim3; k++)
            x[i][j][k] = i+j+k;

for(i=0; i<dim1; i++) free(x[i]);
free(x); free(temp);
```

Uses of pointers

5. Calling C from R

When you call a C function from R, everything is a pointer. We’ll look at how to do that later.
Care with pointers

- Don’t access/modify areas of memory that you don’t have rights to.
- Be careful about going off the ends of arrays.
- Check that calloc/malloc worked.
- Free allocated memory before exiting.

printf control codes

```c
#include <stdio.h> /* prog9.c */
double x;

printf("--> |%d|--", 123); --> 123|--
printf("--> |%5d|--", 123); --> 123|--
printf("--> |%-5d|--", 123); --> 123|--
printf("--> |%s|--", "hello"); --> hello|--
printf("--> |%9s|--", "hello"); --> hello|--
printf("--> |%-9s|--", "hello"); --> hello|--

x = -17.89306213;
printf("--> |%f|--", x); --> -17.893062|--
printf("--> |%7.2f|--", x); --> -17.89|--
printf("--> |%e|--", x); --> -1.789306e+01|--
printf("--> |%g|--", x); --> -17.8931|--
```