C


```
/* A simple example program
 *
 * Written for CSC 228
 * Sept. 11, 1995
 */

#include <stdio.h>

main ()
{
    printf ("Hello, world!\n");
}
```


Declarations

Example:

```
int i;
double d = 5;
int a[100]; /* an array */
char s[21]; /* a string */
```

Syntax:

```
<type> <variable name>, ... [= <value>];
```

Rules:

- Declarations must precede executable statements
- Arrays are always index from 0 -- the array a[100] has 100 elements numbered from 0 to 99

Changing Variable Values

Example:

```
int x, y;
x = 2;
y = 4 + 5;
```

Syntax:

```
<variable name> = <expression>;
```

Rules:

- no colon before the equal sign
- semicolon at end

Incrementing a Variable

Example:

```
x = x + 1;
x += 1; /* Turing copied this from C */
```

Getting Fancy

```
x++;
++x;
y = x++;
z = ++y;
```

Don't use these until you have more experience!

Conditional Statements

Example:

```
if (age < 0)
{
   printf ("warning: negative age\n");
   age = -age;
}
```

Rules:

- parentheses required around conditional expression
- no "then"
- no "end if"

Note:

- **may** use curly braces when there is only one line in body
- **must** use curly braces when there is more than one line in body

More Conditionals

Example:

```
if (x < 0)
   printf ("x is less than 0\n");
else if (x == 0)
   printf ("x is equal to 0\n");
else
   printf ("x is greater than 0\n");
```

Example:

```
if (x < 0)
   if (y < z)
      printf ("y is less than z\n");
else
   printf ("x not less than 0\n");
```

Wrong! The "else" belongs to the inner "if"

Strings

C has no strings. Instead, we use arrays of char:

```
char s[3] = "hi"; /* invisible '\0' (NUL) */
char t[] = "there";
   /* computer figures out length */
```

- concatenation
  ```
  strcat (s, t); /* attach t to end of s */
  ```
- length
  ```
  strlen ("there"); /* returns 5 */
  ```
- copying
  ```
  strcpy (s, t); /* copy t on top of s */
  ```
- comparison
  ```
  strcmp (s, t) /* sgn (s - t) */
  ```
- substring:

  No such thing. Use a loop, treating string as an array

Booleans

C has no boolean type. Instead, we use macros:

```
#define BOOLEAN int
#define TRUE 1
#define FALSE 0
```

Boolean operations:

```
== != && || !
```

Example:

```
BOOLEAN xIsFive, zAsBig;
xIsFive = (x == 5);
zAsBig = (z >= x);
if (xIsFive || zAsBig) {
    ...
}
```


While Loops

Example:

```
/* print “hi” forever */
while (1)
   printf (“hi”);
```

Syntax:

```
while (<condition>)
   <statement>
```

Example:

```
/* count 1, 2, 3, ... */
int i = 0;
while (1)
{
   i += 1;
   printf (“%d\n”, i);
}
```


For Loops

Example:

```
/* print “hi” three times */
int i; /* i continues to exist when loop ends */
for (i = 0; i < 3 ; i++)
   printf (“hi”);
```

Syntax:

```
for (statements1; condition; statement2)
   statement3;
```

Equivalent to:

```
statements1;
while (condition) {
   statement3;
   statements2;
}
```


Output

Example:

```
#include <stdio.h>

int i = 10;
float f = 2.5;
double d = 3.1415926539;
char s[] = “hi”;

printf (“integer i = %d\n”, i);
printf (“float f = %f\n”, f);
printf (“double d = %lf\n”, d); /* long float */
printf (“string s = %s\n”, s);
printf (“all: %d %f %lf %s”, i, f, d, s);
```

Input

Example:

```
#include <stdio.h>

int i;
float f;
double d;
char s[10];

scanf ("%d", &i); /* &i is a pointer to i */
scanf ("%f", &f);
scanf ("%lf", &d);
scanf ("%s", s); /* why no "&" ? */

printf ("all: %d %f %lf %s", i, f, d, s);

scanf ("%d %f %lf %s", &i, &f, &d, s);
```


Input/Output

```
/*
 * Echo standard input to standard output.
 */

#include <stdio.h>

main ()
{

   int c;

   while ((c = getchar ()) != EOF)
      putchar (c);
}
```


Pointers

C relies heavily on pointers:

```
int i;
int *pi; /* *pi is an int so pi is a pointer */
int a[100];

pi = &i; /* pointer to i */
*pi = 3; /* same as i = 3 */
```

An array name by itself is a pointer to the first element:

```
*a = 5; /* same as a[0] = 5 */
*(a + 3) = 10; /* same as a[3] = 10; */
/* Note: *a + 3 means something else */
```

Since strings are arrays of characters, strings are pointers.

```
strcpy (s, "hi"); /* same as following line */
s[0] = 'h'; s[1] = 'i'; s[2] = '\0';
/* Note: s = "hi" is wrong! */
```


Functions

```
int incr (int i);
/*
 * Return i incremented by 1:
 * incr takes an integer argument and
 * returns an integer
 */
{
   int j;

   j = i + 1;
   return j;
}

main ()
{
    int k, m = 4;
    m = incr(k);
    printf ("k = %d, m = %d\n", k, m);
}
```

More about Functions

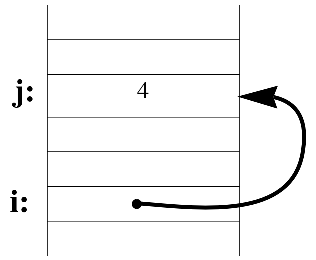
C has no procedures or “var” parameters.

Example:

```
void incr (int i);
/*
 * increment i by 1
 * Does NOT work!
 */
{
    i = i + 1;
}
```


Make-your-own Var

```
void incr (int *i);
/*
 * increment i by 1
 */
{
    *i = *i + 1;
}

main()
{
    int j;
    j = 4;
    incr (&j);
    printf ("j = %d\n", j);
}
```



`&j` and `i` are pointers to the same integer: `incr` can change the integer (side effect), but not the pointer

Make-your-own Type Name

Examples:

```
typedef char *string_t;
typedef int boolean_t;

string_t msg = "No more room";
boolean_t class_is_full;

if (num_students >= MAX_SIZE)
     class_is_full = TRUE;
```

Comments:

- typedef does not create a new type, only a new label for an existing type
- typedefs make your code easier to read

Structures

struct **can be used to create or define a new type:**

```
struct str_data {
     char *string;
     int length;
};
struct str_data s1, s2;
```

A member of a structure is referred to by:

```
<structure name>.<member name>
```

Example:

```
/* dynamically allocate 80 bytes of memory */
s1.string = (char *) malloc(80);

strcpy (s1.string, "How now brown cow.");
s1.length = strlen(s1.string);
s2 = s1; /* can copy structures, pass to fcns */
```

Arrays of Structures

Example:

```
#define MAX_ELEMENTS 100

typdef struct {
     char *string;
     int length;
} my_string_t;

/* now create a list of MAX_ELEMENTS elements
 * of type my_string_t
 */
my_string_t list[MAX_ELEMENTS];

for (i = 0; i < MAX_ELEMENTS; i++) {
    list[i].string = (char *) malloc(80);
    gets(list[i].string);
    list[i].length = strlen(list[i].string);
}
```


Pointers to Structures

These are all equivalent:

```
list[0].length = ...
(*list).length = ...
list->length = ...
```

Rules:

- **p->y** is shorthand for **(*p).y**
- means “follow the pointer **p** into structure member **y**
- it is more efficient to pass pointers to structs rather than the structs themselves

Self-Referential Structures

Example: Linked list

```
typedef struct element {
     int data;
     struct element *next;
} node_t;
```

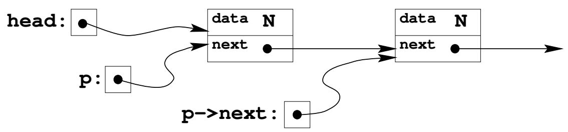
Rules:

- a structure cannot contain itself, but it can contain a pointer to itself
- **next** is a pointer to a **node_t** structure
- we can use this to define a pointer to the head of the list:
  node_t *head = NULL;

Note: a NULL pointer is used to mark the end of the list

Traversing a Linked List

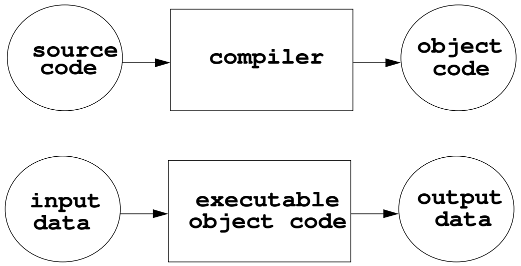
```
node_t *p;

p = head;
while (p != NULL) {
     printf (“data = %d\n”, p->data);
     p = p->next;
}
```



Compiling

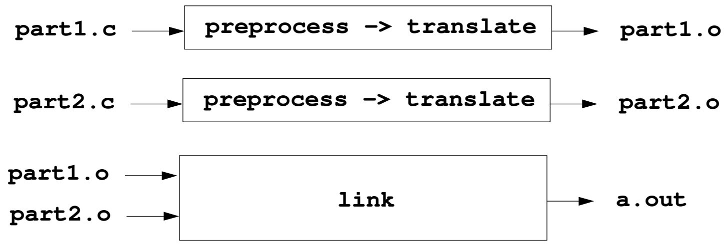
Steps to a running program:

```
% vi prog.c
% gcc prog.c
% a.out
```



Separate Compilation

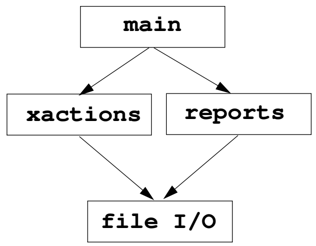
```
% gcc -c part1.c
% gcc -c part2.c
% gcc part1.o part2.o
% a.out
```



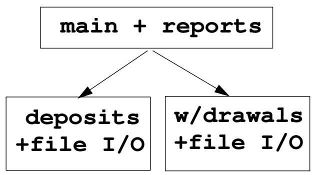
Makefiles

```
# Sample Makefile from ~jer/C/sepcomp

# The following two lines say that pgm
# depends on prog.o and sub.o.
# The action required to create pgm is
# to link prog.o and sub.o.

pgm: prog.o sub.o
    gcc -o pgm prog.o sub.o

# The next four lines specify what each of
# prog.o and sub.o depend upon, and how these
# object files are to be created.

prog.o: incl.h prog.c
    gcc -c prog.c
sub.o: incl.h sub.c
    gcc -c sub.c
```


Modular Programming



Good Style



Poor Style

Header Files

```
/* File: myheader.h
 * Definitions for program build.c
 */

/* Booleans */
 typedef int boolean_t;
 #define TRUE 1
 #define FALSE 0

/* node type */
typdef struct node_t {
  ...
}

/* function prototypes */
 node_t *NewNode(char *label, int cost);
 void KillNode (node_t *node);
```


Naming Conventions

Consistency is bliss. A suggested format to follow:

Constants: use all upper case

```
#define TRUE 1
#define MAX_ELEMENTS 50
```

Variables: use all lower case

```
char *name; int num_students;
```

Functions: use upper case for first letter of each word

```
float *GetCompressedValue(); char *GetNextWord();
```

Types: use all lower case with a "_t" suffix (_t == type)

```
typedef struct node_t {...}
```


Programming Errors I

```
#include <stdio.h>

main ()
{
     int i;

     /* echo one number */
     scanf ("%d", i);                     <- &i
     printf ("input = %d\n", i);
}
```

Must provide the address of an integer if the function is going to assign a value to it.

Programming Errors II

```
#include <stdio.h>

main ()
{
     char c;      <- int c;

     while ((c = getchar ()) != EOF)
        putchar (c);
}
```

getchar() returns an integer

- might not be able to compare EOF correctly against a character value

Programming Errors III

```
#include <stdio.h>

main ()
{
     int i;

     scanf ("%d", &i);
     if (i = 0)                <- i == 0
        puts ("false");
     else
        puts ("true");
}
```

The assignment, i = 0, will return a value of 0, so the program will always output "true"

Programming Errors IV

```
#include <stdio.h>

main ()
{
     char *pc;          <- char pc[100];

     scanf ("%s", pc);
     printf ("%s\n", pc);
}
```

***pc is an uninitialized pointer so results are unpredictable**

Programming Errors V

```
#include <stdio.h>

main ()
{
     char s[] = "hi";     s contains string "hi"
     char *pc = "hi";

     if (pc == "hi")      pc points to memory
        puts ("yes");     location "hi"
     else                 --> outputs "yes"
        puts ("no");

     pc = s;              pc now points to s,
     if (pc == "hi")      which is a different
        puts ("yes");     memory location from
     else                 "hi" --> outputs "no"
        puts ("no");
}
```


Finding Bugs

Insert "print" statements:

```
printf ("At point A, %s = %d\n", "x", x);
```

Use a debugger to help you catch bugs in the act:

```
% gdb prog
GDB is free software and you are welcome to ...
(gdb) run
```

The GNU Debugger

With GDB (The GNU Debugger) you can perform 3 main debugging functions:

- Start your program, specifying anything that might affect its behaviour.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped or crashed.

Using GDB

Compile your program with the -g option:

```
% gcc -g -o prog prog.c
```

To run your program in GDB:

```
% gdb prog
(gdb) run
```

To examine your program after a crash (core dump):

```
% prog
Segmentation fault (core dumped)

% gdb prog core
Core was generated by `prog'.
Program terminated with signal 11, Segmentation
fault.
(gdb) ...
```


Some GDB Commands

Set a breakpoint:

```
(gdb) break [file:]function
```

Run your program:

```
(gdb) run [arglist]
```

List 10 lines of code:

```
(gdb) list
```

Display the value of an expression:

```
(gdb) print expr
```

Display a value after each stoppage:

```
(gdb) display expr
```


More GDB Commands

Execute next line stepping <u>over</u> function calls:

```
(gdb) next
```

Execute next line stepping <u>into</u> function calls:

```
(gdb) step
```

Continue after a break:

```
(gdb) c
```

Getting help:

```
(gdb) help name
```

Exiting:

```
(gdb) quit
```

Debugging A Program I

Example: (searching for a value in a linked list)

```
node_t *Search(node_t *head, int data)
/* search for data, return pointer to the node */
{
   node_t *p;

   p = head;
   do {
      if (p->data = data)
         return (p);
      else
         p = p->next;
   } while (p != NULL);
   return (NULL);
}
```


Debugging A Program II

There is a bug:

Input:

```
1 2 3 4 -1  (values for linked list)
17          (value to search for)
```

Output:

```
17 is found  (! but it's not in the list)
17 3 2 1     (! 4 is missing)
```

We know the program worked before adding the Search function, so it must be a bug in the search code.

Steps to find the bug:

- start up GDB:

```
% gdb linklist
```


Debugging A Program III

- set a breakpoint at the Search function:

```
(gdb) break Search
Breakpoint 1 at 0x108e8: file linklist.c,line 48.
(gdb)
```

- run the program and provide some input:

```
(gdb) run
Starting program: linklist
1 2 3 4 -1 17
Breakpoint 1, Search (head=0x20d58, data=17)
51          p = head;
(gdb)
```

- display some variables:

```
(gdb) display p
1: p = (node_t *) 0x20d58
(gdb) display *p
2: *p = {data = 4, next = 0x20d48}
(gdb)
```


Debugging A Program IV

- step through the code and watch the variables change:

```
(gdb) step
53             if (p->data = data)
2: *p = {data = 4, next = 0x20d48}
1: p = (node_t *) 0x20d58

(gdb) step
54                return (p);
2: *p = {data = 17, next = 0x20d48}
1: p = (node_t *) 0x20d58

(gdb) step
59       }
2: *p = {data = 17, next = 0x20d48}
1: p = (node_t *) 0x20d58
```

We found the bug:

After line 53 executes, **`p->data`** has changed to 17! Why?

Debugging A Program V

But there is another bug:

Input:

```
-1    (empty list)
17    (value to search for)
```

Output:

```
Segmentation fault (core dumped)
```

See if you can find the bug. Start with:

```
% gdb linklist core
```


MXGDB: Graphical Interface to GDB