Testfile

COLLABORATORS

	TITLE : Testfile	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		November 18, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Testfile

1.1 Table Of Contents

```
TABLE OF CONTENTS


auto
break
case
char
const
continue
default
do
double
else
enum
extern
float
for
goto
if
int
long
register
return
short
signed
sizeof
static
struct
switch
typedef
union
unsigned
void
volatile
while
```

1.2 "

```
NAME
   auto - Is an access modifier which declares variables to be
          local.

SYNOPSIS
   auto <datatype> <variable list>;

FUNCTION
   Auto declares variables to be local.  It is almost never used
   by anyone.  It exists only because it was part of the original
   C set of keywords.  To declare a local variable all you need to
   do is to declare it inside the code block that you intend to
   use it in.  A code block begins at the start of a curly brace
   "{" and ends with another curly brace "}".

INPUTS
   <datatype> - A basic C type or a user defined (complex) type.
   <variable list> - A list of variables separated by commas.

RESULT
   none.

EXAMPLE
   auto int x, y, z = 25;

NOTES
   Never use, it just doesn't look very professional.

BUGS
   none. - If there are, there is something really wrong with your
           compiler;-)

SEE ALSO
   extern, static, register.

   void, char, int, float, double,
   long, short, signed, unsigned,
   const, volatile.
```

1.3 /break

```
NAME
   break - Used either as a part of a switch statement, or to
           prematurely break out of a loop.

SYNOPSIS
   switch (<var>)
   {
      case <const1>:
         <statement sequence>;
         break;
      case <const2>:
```

```
         <statement sequence>;
         break;
      default:
         <statement sequence>;
   }

   or

   while (<var>==0)
   {
      if (<var>==0) break;
      printf("This text will not be printed\n");
   }
```

FUNCTION

This C keyword will prematurely break you out of a looping structure or can be used to terminate a statement sequence after a case statement. In this last role it is optional but is almost always used unless there is a specific reason not to.

INPUTS

none.

RESULT

none.

EXAMPLE

```
   switch (x)
   {
      case 1:
         printf("x is equal to 1.\n");
         break;
      case 2:
         printf("x is equal to 2\n");
         printf("and it is not equal to 1.\n");
         break;
      default:
         printf("x is not equal to 1 or 2\n");
         printf("it must be something else then.\n");
   }

   or

   while (x==0)
   {
      if (x==0) break;
      printf("This text will not be printed\n");
   }
```

NOTES

Switch statements do not require the break statement, but in most instances you will want to use break. An example of when you wouldn't want to use break would be if you wanted the first case <statement sequence> to execute and also the second <statement sequence> if the first is called

```
   but only the second if the second case is called.

BUGS
   none - If you're even considering that break might have a
          bug, maybe you should either rethink the problem, or
          invest in an expensive compiler, only to find that
          the problem is still there.

SEE ALSO
   switch, continue.
   case, for, while, do.
```

1.4 /case

```
NAME
   case - Used with the switch statement

SYNOPSIS
   case <const>: <statement sequence>;

FUNCTION
   Case is used with the switch statement and can be considered
   to be part of the switch statement as "do" is associated with
   a "do while" loop.

INPUTS
   <const> - Any integer constant, can be either declared with
             the "const" keyword or simply typed in directly as
             an integer number.  Character constants and
             enumerations may be used also.
   <statement sequence> - The single statement or block of
                          statements which is/are to be executed
                          by default.

RESULT
   none.

EXAMPLE
   switch (x)
   {
      case 1:
         printf("x is equal to 1.\n");
         break;
      case 2:
         printf("x is equal to 2\n");
         printf("and it is not equal to 1.\n");
         break;
      default:
         printf("x is not equal to 1 or 2\n");
         printf("it must be something else then.\n");
   }

NOTES
   It should be noted that in a switch statement any constant of
   type char, (or any other type other than an int), will first
```

```
   be converted to an integer.  This doesn't really matter as
   far as the programmer is concerned unless the variable type
   is larger than an integer, in which case the value will get
   garbled and the program will behave in a way which was
   unintended.  The reason switch will only handle integral types
   is for speed reasons.

BUGS
   As stated above, only works with types smaller or equal in
   size to integers.

SEE ALSO
   switch, break, const, int, char.
```

1.5 /char

```
NAME
   char - Is a data type which declares a variable of type char.

SYNOPSIS
   char <variable list>;

FUNCTION
   Char declares a variable of type char, meaning that the
   variable will hold a single ASCII character.  Technically
   type char is an 8 bit type that will hold any number between
   -128 and 127.  Often operating systems will declare their own
   variable types and will usually have one called BYTE or UBYTE
   or the like which is either a char or unsigned char.

INPUTS
   <variable list> - A list of the variables you are declaring,
                     separated by commas.

RESULT
   none.

EXAMPLE
   char x, y = 'a', z[257] = "This is a string of characters";

NOTES
   To store a string of characters in a variable,
   (i.e. Hello there) you make an array of characters and store
   each letter in one element of that array.  The declaration
   would look like this, (char thsisastrn[257];)  You could
   then store a sentence of up to 256 characters in length.
   Also I'd like to mention that there is a w_char type that isn't
   a built in C type but is defined in one of the standard headers
   and can be used to store characters for languages besides
   english and the like.  I believe that w_char is usually used
   for unicode, but I'm not really sure about the details.  If
   your interested in making your programs portable to other
   countries and cultures then check it out.  Also it should be
   noted that in C++ w_char is a built in type.
```

```
BUGS
   none. - Man, if you've got bugs with this, you've got some
           serious problems!!!

SEE ALSO
   void, int, float, double.

   long, short, signed, unsigned,
   const, volatile,
   extern, static, register, auto,
   struct, union, typedef, enum.
```

1.6 /const

```
NAME
   const - Is an access modifier which is used to declare a
           special "constant" variable type.

SYNOPSIS
   const <datatype> <name> = <expression>, <name> = <expression>,
   etc, etc;

FUNCTION
   Const declares and initializes variables which cannot be
   altered by the program after initialization.  The declaration
   looks the same as a normal variable declaration with the
   exception that the "const" keyword precedes it, and each
   constant must be initialized at the time it is declared, (i.e.
   given a value such as x = 25).  An example of a practical use
   would be to make PI a const float equal to 3.14...

INPUTS
   <datatype> - A basic C datatype, (i.e. char, int, float, double),
                or a user defined (complex) data type.
   <name> - Name of the constant being declared.
   <expression> - Any "statement" which evaluates to  the
                  declared type, (i.e. For type int, 25 would be
                  acceptable).

RESULT
   none.

EXAMPLE
   const float PI = 3.14, e_raised2_x = 2.71828;

NOTES
   While const types may not be altered by the program, they may
   be altered by some hardware dependent means.  A good example of
   this would be to store the address of a place in memory that
   always contains the current time in a constant pointer that
   your program can't alter, but can look at whenever it wants
   to know the time.  Honestly a technique like this might make
   more sense in C++ where you can declare variable in the middle
   of your code.  Also in this example you should probably use
   the keyword volatile to keep the compiler from doing any
```

```
   optimizations that would cause flaky results.  I was really
   just giving it as a for instance so you can take it or leave
   it.  Also, the word "constant" refers to both the special type
   and any fixed value in the program,
   (i.e. 25+25 or "Hello world\n").  The reason for this is that
   from the compiler's point of view they are stored in the same
   way.  Also, you can think of it from the standpoint that 25+25
   is a constant expression because the answer will always be 50.
   In many cases, (OK ALL CASES), you may use a #define
   preprocessor directive in place of the const variable type.
   It should be noted, however, that #define works in a different
   manner than const does, (at least from the compilers point of
   view.)

BUGS
   none.

SEE ALSO
   volatile, #define.
   void, char, int, float, double,
   long, short, signed, unsigned.

   extern, static, register, auto,
   struct, union, typedef, enum.
```

1.7 /continue

```
NAME
   continue - Used in a loop to force the next iteration, (cycle),
              immediately.

SYNOPSIS
   for (<var>=0; <var><100; <var>++)
   {
      if (<var>>=50) continue;
      printf("%d\n", <var>);
   }

FUNCTION
   Continue is used to force the next iteration of a looping
   structure, (i.e. for, while, do while).  Continue is used in
   much the same manner as the break statement, except that
   instead of breaking completely out of the loop, it just causes
   the loop to skip to the end, and then start on the next
   iteration.  In the examples for instance, only the numbers
   0-49 are printed even though the loop is executed and the <var>
   is incremented 100 times.

INPUTS
   none.

RESULT
   none.

EXAMPLE
```

```
   for (x = 0; x<100; x++)
   {
      if( x>=50) continue;
      printf("%d\n", x);
   }

NOTES
   When continue is used in a for loop, the variable is
   incremented, then the conditional statement is tested, then the
   loop starts again.  This is the same action which would take
   place if the end of the code block had been reached.  In
   "while" and "do while", only the conditional statement is
   evaluated, (as they do not have the built in ability to
   increment variables), and then the loop starts again.

BUGS
   none.

SEE ALSO
   break.
   for, while, do.
```

1.8 /default

```
NAME
   default - Used in conjunction with the switch statement to
             specify a default course of action.

SYNOPSIS
   default: <statement sequence>;

FUNCTION
   Default is used in conjunction with the switch statement to
   specify a default course of action when no case statements
   have been matched.  In essence, it is to switch, what else is
   to if.

INPUTS
   <statement sequence> - The single statement or block of
                          statements which is/are to be executed
                          by default.

RESULT
   none.

EXAMPLE
   switch (x)
   {
      case 1:
         printf("x is equal to 1.\n");
         break;
      case 2:
         printf("x is equal to 2\n");
         printf("and it is not equal to 1.\n");
         break;
```

```
      default:
         printf("x is not equal to 1 or 2\n");
         printf("it must be something else then.\n");
   }

NOTES
   none.

BUGS
   none.

SEE ALSO
   switch, case.
```

1.9 /do

```
NAME
   do - Part of the "do while" looping construct.

SYNOPSIS
   do
   {
      <statement sequence>;
   } while (<conditional statement>);

FUNCTION
   Do is used to start a "do while" loop.  A "do while" loop
   behaves in almost the same manner as a while loop, with the
   exception that the <statement sequence> is executed before
   the <conditional statement>.  This has the effect that the
   loop always executes at least once regardless of whether the
   <conditional statement> is true or false.

INPUTS
   <statement sequence> - The single statement or block of
                          statements which is/are to be executed.
   <conditional statement> - A statement which evaluates to either
                             true or false.  False is equal to 0
                             and true is anything else, however
                             it is common practice to have -1 or 1
                             represent true.

RESULT
   none.

EXAMPLE
   do
   {
      printf("Enter a number between 1 & 5...\n");
      scanf("%d", &x);
   } while ((x<1) || (x>5));

NOTES
   In the example above the loop reads in x from the user and then
   tests to see if that value is acceptable.  A "while" loop could
```

```
   not have been used effectively because x could have been set to
   anything, including a value which would have caused the
   program to skip the loop entirely.  You could conceivably
   set x to zero before you enter a "while" loop, but it is
   easier and more readable simply to use "do while".  Nuff said.

BUGS
   none.

SEE ALSO
   while, for.
   if, switch.
```

1.10 /double

```
NAME
   double - Is a data type and keyword used to declare variables
            which hold double precision floating point values.

SYNOPSIS
   double <variable list>;

FUNCTION
   Double declares variables that can be used to store double
   precision floating point numbers.  A double precision number
   is 64 bits wide in memory, meaning that the number can
   range from -1.7e-308 to 1.7e308.  It should be noted that
   unlike int, but like float, the double type can be used to
   store fractional numbers which are extremely small or extremely
   large.  Double should be used when variables of type float are
   too small to hold the desired number or when extreme accuracy
   in calculations is needed.

INPUTS
   <variable list> - A list of the variables you are declaring,
                     separated by commas.

RESULT
   none.

EXAMPLE
   double x, y, z = 3.14;

NOTES
   In older versions of C, double was a synonym for long float.
   As per the ANSI standard, long float is no longer accepted and
   the double keyword must be used.  Both float and double require
   more space and more CPU time to do calculations on than do
   ints.  Try to avoid using them inside of loops that are crucial
   to the speed of the program.  Instead try to put them in places
   where the expression in question only gets executed every now
   and then, and not 100 times a second.  There is a way to do
   division on ints so that the remainder is known.  This is much
   faster and more information is available, (see div).  Many
   times you must link a special library to your code to do
```

```
   floating point math, (i.e. to be able to use floats and
   doubles).  This is because they are so much slower and most
   programmers try to avoid them except in special cases.  It is
   also because there are different ways for the compiler to do
   the math internally.  If you have an FPU for example, you can
   compile your code to take advantage of that special hardware by
   linking it to a library that supports an FPU, or you can link
   to a generic library, (mieee library for example), and all of
   the calculations will be preformed by the software, with the
   effect that your program will run on computers that aren't
   lucky enough to have an FPU, (like my computer for example).
   If in doubt, use mieee, it's safer.  When I speak of libraries
   it should be noted that I am talking about link libraries which
   come with your compiler and not the amigados shared libraries
   which reside in the libs: assign.

BUGS
   Slow.

SEE ALSO
   void, int, char, double.

   long, short, signed, unsigned,
   const, volatile,
   extern, static, register, auto,
   struct, union, typedef, enum.
```

1.11 /else

```
NAME
   else - Part of the if statement which specifies a default
          course of action, (OPTIONAL).

SYNOPSIS
   if (<conditional statement>)
   {
      <statement sequence>;
   }
   else
   {
      <statement sequence>;
   }

FUNCTION
   Else specifies a default course of action to take when the
   <conditional statement> evaluates to false.  If it evaluates
   to true then else is skipped.

INPUTS
   <conditional statement> - A statement which evaluates to either
                             true or false.  False is equal to 0
                             and true is anything else, however
                             it is common practice to have -1 or 1
                             represent true.
   <statement sequence> - The single statement or block of
```

```
                          statements which is/are to be executed
                          by default.

RESULT
   none.

EXAMPLE
   if (x)
   {
      printf("x is unequal to zero.\n");
   }
   else
   {
      printf("x is 0.\n");
   }

NOTES
   Else can be used in a couple of ways.  It can be used as shown
   above to produce an either or type of decision or can be used
   to string together several ifs, (i.e.

   if (x==0) {<statement sequence>;}
   else if (x<0) {<statement sequence>;}
   else if (x>0) {<statement sequence>;}).

   You could just use 3 ifs in this situation, however the
   "if else if ladder" executes faster because if the first
   expression is met then the rest are skipped by default and
   don't waste time evaluating to false.

BUGS
   none.

SEE ALSO
   if, switch.
   for, while, do.
```

1.12 /enum

```
NAME
   enum - Defines an enumeration data type, and is used to declare
          variables of that type.

SYNOPSIS
   enum <tagname> {<enumeration list>} <variable list>;

FUNCTION
   Enum defines a new data type, much the same way struct and
   union do.  These new data types that the user creates are
   often referred to as "user defined types".  Enum is also used
   in the declaration of variables that are to be of a type
   which was defined using enum.  Enum really can't be
   explained without an example so we'll refer to the example
   below.  "coins" is the new data type.  "money" is a global
   variable of type coins.  "moremoney" is a local variable of
```

```
   type coins.  Now that we have our two variables "money" and
   "moremoney" we can assign them a value of type coins, (i.e.
   any of {penny, nickel, dime, or quarter}.  Conditional tests
   may be performed on "money" and "moremoney".  We used switch
   below, but you may use if statements, while, for, do while,
   or any other C construct that allows conditional (true/false)
   tests.  I think if you look over the example, you'll get the
   idea of what enumerations are good for.

INPUTS
   <tagname> - The name of the variable type being created, (i.e.
               you would use this name when declaring variables
               of this type.
   <enumeration list> - The list of enumeration names that will
                        be used as the "pseudo data" for any
                        variable being declared as type <tagname>.
   <variable list> - A list of the variables you would like to
                     declare separated by commas.  This allows
                     variables to be declared at the same time
                     the data type is defined.

RESULT
   none.

EXAMPLE
   enum coins {penny, nickel, dime, quarter} money;

   int main(void)
   {
      enum coins moremoney;

      money = penny;
      moremoney = dime;

      switch (money)
      {
         case penny: printf("Hey I've got a penny.\n");
         break;
         case nickel: printf("Hey I've got a nickle.\n");
         break;
         case dime: printf("Hey I've got a dime.\n");
         break;
         case quarter: printf("Hey I've got a quarter.\n");
      }

      switch (moremoney)
      {
         case penny: printf("Hey, I've got another penny.\n");
         break;
         case nickel: printf("Hey, I've got another nickel.\n");
         break;
         case dime: printf("Hey, I've got another dime.\n");
         break;
         case quarter: printf("Hey, I've got another quarter.\n");
      }

      exit(0);
```

```
   }


NOTES
   It should be noted that enum is just a fancy way of disguising
   the int variable type.  In the example above "money = penny" is
   equivalent to "money = 0", and "moremoney = dime" is equal to
   "moremoney = 2".  The <enumeration list> assigns the different
   names integer values beginning with 0 and going up by one, (i.e.
   1, 2, 3, 4, 5, 6, etc).  You can alter this pattern by using
   an assignment in the definition, (i.e.  enum coins {penny = 0,
   nickel = 5, dime = 10, quarter = 25}.  By doing this you could
   create tests like ( if (nickle+nickle==dime) printf("2 nickels
   equals 1 dime.");).  You can even assign two names the same
   value, (i.e. enum coins {penny, nickel, dime, quarter,
   two_bits = 3, half_dollar}.  In this situation penny = 0,
   nickel = 1, dime = 2, quarter = 3, two_bits = 3, and
   half_dollar = 4.

BUGS
   Any restrictions that would apply to type int, also apply to
   enums, on the other hand any advantages that ints have over
   other types also apply to enums.  This is why enumerations
   work with the switch statement so well.

SEE ALSO
   struct, union, typedef.

   switch, if,
   for, while, do.
```

1.13 /extern

```
NAME
   extern - Is an access modifier used in variable declarations
            to specify that the variable is declared in another
            source file which will be linked later.

SYNOPSIS
   extern <data type> <variable list>;

FUNCTION
   Extern is used in variable declarations to specify that the
   variable is declared in another source file which will be
   linked later.  It isn't a definition per say, but merely lets
   the compiler know that the variable is out there.  If you don't
   tell the compiler that the variable is declared elsewhere
   then it will most likely spit up an "unknown identifier" error,
   and if you don't use the extern keyword then the linker will
   spit out a "multiply declared identifier" error or something
   like that.  Basically, a program can't define something
   twice, so you have to use extern to tell the compiler that even
   though this variable hasn't been declared yet, it can still go
   ahead and compile the proggy just as if it had been declared.
```

```
INPUTS
   <data type> - Is any one of C's built in data types like: int,
                 char, float, double, etc.  Also you may use your
                 own user defined (complex) data types,
                 (i.e., structs, unions, etc).
   <variable list> - Is a list of variables that have been
                     declared elsewhere, separated by commas.  see
                     the example for clarification.

RESULT
   none.

EXAMPLE
   /* sourcenumber1.c */
   void testfunction(void);

   int x, y = 2, z[50] = {0};

   int main(void)
   {
      testfunction();

      return 0;
   }

   /* sourcenumber2.c */
   extern int x, y, z[50];

   void testfunction(void)
   {
      printf("%d, %d, %d\n", x, y, z[0]);
   }

NOTES
   The example above is cut up into two source files.  Something
   to note about using extern is that it only makes sense to use
   it when dealing with multiple source code files.  Also notice
   how the arrays were handled.  Typically I like to declare
   something as external exactly the same way I declared it
   originally, (but without the initialization).  In the second
   source file I could just as easily have written,
   (extern x, y, z[];).  Stick to whatever makes the most sense
   to you, I just find it easier to mimic the original form as
   it's less semantic bs to remember and I think it aids in
   readability.

BUGS
   I certainly hope not!!!

SEE ALSO
   volatile, register,
   const, signed, unsigned, short, long.
```

1.14 /float

```
NAME
   float - Is a data type used for holding floating point numbers
           between -3.4e38 and 3.4e38.

SYNOPSIS
   float <variable list>;

FUNCTION
   float is a data type used for holding floating point numbers
   between -3.4e38 and 3.4e38.  A floating point number takes more
   time for the computer to do calculations on than int.  It is
   always preferable to use int if you can use one type or the
   other however there are just some spots that you can't get out
   of using floating point numbers.  Floating point numbers are
   useful in two common happenstances: one, the number being
   stored is of exceptional accuracy, (like .000231), and
   maintaining that accuracy is crucial, or two, if the number is
   too large to store in an int or a long int,
   (like 3.2861 * 10^28).

INPUTS
   <variable list> - A list of the variables being declared
                     separated by commas.

RESULT
   none.

EXAMPLE
   float x, y = 3.14, z[5] = {2.2, 3.3, 4.4, 5.5, 6.6};

NOTES
   When you do decide you need floats, try to avoid using them
   inside of loops that that are crucial to the speed of the
   program.  Instead try to put them in places where the
   expression in question only gets executed every now and then,
   and not 100 times in a second.  There is a way to do
   division on ints so that the remainder is known.  This is much
   faster and more information is available, (see div).  Many
   times you must link a special library to your code to do
   floating point math, (i.e. to be able to use floats and
   doubles).  This is because they are so much slower and most
   programmers try to avoid them except in special cases.  It is
   also because there are different ways for the compiler to do
   the math internally.  If you have an FPU for example, you can
   compile your code to take advantage of that special hardware by
   linking it to a library that supports an FPU, or you can link
   to a generic library, (mieee library for example), and all of
   the calculations will be preformed by the software, with the
   effect that your program will run on computers that aren't
   lucky enough to have an FPU, (like my computer for example).
   If in doubt, use mieee, it's safer.  When I speak of libraries
   it should be noted that I am talking about link libraries which
   come with your compiler and not the amigados shared libraries
   which reside in the libs: assign.
```

```
BUGS
   Contact the manufacturer of your compiler on this one, they've
   obviously been exposed to too many drugs in the 60's.

SEE ALSO
   void, int, char, double.

   long, short, signed, unsigned,
   const, volatile,
   extern, static, register, auto,
   struct, union, typedef, enum.
```

1.15 /for

```
NAME
   for - The C keyword used for constructing, (most notably), a
         for loop.  The for keyword is very versatile however, and
         can be used to construct different kinds of loops,
         (like infinite loops, and pseudo while loops).

SYNOPSIS
   for (<variable assignments>; <test conditions>; <statements>)
   {
      <statement sequence>;
   }

FUNCTION
   The for loop is useful for keeping track of how many iterations
   have gone by during an operation, and also to terminate based
   on either the number iterations, or on some other unrelated
   criteria.  C for loops have much more bite to them then other
   languages.  A for loop is really just a while loop with added
   capabilities.  The statement for (;x<10;) printf("Hi\n"); is
   exactly equivalent to while (x<10) printf("Hi\n");, (Well at
   least functionally it is, I don't know how the computer sees
   it).  All of the parts of the for loop are completely optional
   so you can pick and choose what you need and what you don't.
   for example the statement for (;;) {  }  throws the program
   into an infinite loop, which may only be broken out of
   using a break statement.  Also it is worthwhile to mention
   that each part of the for loop may contain multiple
   declarations, test conditions, and statements. for instance
   for (x = 0, y = 10; x<=10 && y>=0; x++, y--)
      printf("%3d, %3d\n", x, y);
   tells the program to set x equal to 0 and y to 10, to keep
   going until either x is greater than 10 or y becomes less than
   zero and to increment x and decrement y after printf() prints
   it's little message.

INPUTS
   <variable assignments> - A list of variables to assign a value
                            to, or in rare cases any other
                            statement you would wish only to
                            execute at the beginning of the for
```

```
                              loop.  Frankly statements should
                              probably just come before the for
                              loop as a matter of good form.  Each
                              assignment should be separated by a
                              comma.
   <test conditions> - Truly only one test condition can be put
                       here, however, in practice you may just
                       separate each test condition by a && or
                       || or any other C test operator depending
                       on how you think the thing should work,
                       and thus have multiple test conditions.
   <statements> - I called this one statements for lack of a
                  better word.  This space is typically used for
                  incrementing and/or decrementing the values
                  initialized in the first part of the for loop,
                  however, any statements may go here.  Again
                  I think as a matter of good form you should
                  try to stick to incrementing and decrementing,
                  and if you really have to have some statements
                  execute, then just stick them at the end of the
                  <statement sequence>.
   <statement sequence> - The single statement or block of
                          statements which is/are to be executed.

RESULT
   none.

EXAMPLE
   for (x = 0, y = 0; x < 10; x++, y+=2)
   {
      printf("%3d, %3d\n", x, y);
   }

NOTES
   When I first learned C, coming from BASIC, I thought that C
   for loops were needlessly ambiguous, and generally retarded,
   however later, after a couple books, much tinkering, and a lot
   of, "Ohhhhh, that's why;)", I learned that the small sacrifice
   in straight forwardness yielded 10 times to the functionality
   and applicability of the construct.  I personally regard the C
   for loop as one of the most useful features of the language,
   and it's certainly something that sets it apart from most other
   languages which lock you into a certain way of doing it.
   The way I got through the fact that the C for loop isn't very
   obvious to the beginning programmer, is to invent a dialogue
   for translating what the for loop is doing into English.  When
   you see the statement
   for (x = 0; x < 10; x++) {printf("hi\n");
   you can translate it as , set x equal to 0, while x is less
   than 10 execute printf("hi\n") and increment x, then repeat.
   It's useful to look at the for loop as a while loop with added
   capabilities.  Also remember that any and all parts of a for
   loop are completely optional, and that each part may contain
   multiple statements, test conditions etc.

BUGS
   Bugs you say, nope, none here.
```

```
SEE ALSO
   do, while, switch, break, continue, default, if, else.
```

1.16 /goto

```
NAME
   goto - Jumps to another part of the program.  NEVER USE!!!!
          If your code ever leeks out and other programmers get
          wind, you're reputation will never recover, companies
          won't hire you, you'll be an outcast, shunned by all
          but the most loathsome of BASIC programmers.

SYNOPSIS
   goto <label>


   <label>:

FUNCTION
   Goto has no function, it has no reason to exist, but to satisfy
   those unimaginative BASIC programmers who dare set their
   obviously unadaptable feet into C water just to have them
   ripped off by some shark in the guise of a nerdy keyboard
   jockey.  No, but seriously, goto is regarded by all as bad
   programming technique, and should be avoided at all cost on
   pain of unintelligible source code, both to you and anyone else
   who might have business with it later.  There is only one
   conceivable instance that I'm aware of where goto might be an
   actual benefit to source code readability.  Say you have
   yourself nested in several layers of loops and some condition
   happens, (maybe an error or even some debug code), and you have
   to exit out of all of the loops.  It is inconvenient to use
   breaks, because you have to put one into every loop until your
   safely back into wherever the original calling code was.
   Frankly, it's probably a better idea to rethink the way your
   doing it and try to come up with a way that doesn't involve
   goto unless it's just temporary debug code which will be taken
   out after the bug has been solved.

INPUTS
   <label> - Is the only argument goto takes.  You must put a
             label somewhere in the code of the current function
             you are in.  goto will not find a label if it is
             located in another function.  the label is followed
             by a colon when marking the location, but not when
             used with goto.  If I didn't explain that well see
             the example and it should become clear.

RESULT
   none.

EXAMPLE
   label:
      printf("Hello World ;-)\n");
```

```
   goto label;

NOTES
   A lot of C and C++ programmers truly wonder why goto was
   included in the language when it already had such a rich set of
   control structures?  I personally believe it's because C was
   originally cooked up in the 60s or 70s, or something, when
   basic programming was all the rage.  When people finally
   realized that goto wasn't all it was cracked up to be, it was
   too late to take it out of the language for backward
   compatibility's sake.  That being said I'll make a note here
   that should in no way convince you that using goto is OK.  Goto
   is generally faster than the built in control structures
   because it's tailored to the specific task at hand and doesn't
   generate redundant code.  Also I think each time you use a C
   control stucture there's a certain amount of overhead involved,
   but don't quote me on that.  If you ever look at assembly,
   you'll see the equivilant of gotos all over the place.  Goto
   is alot closer to how the machine thinks than the other control
   structures.  However, truly the amount of time lost by using C
   constructs is truly minimal and is barely worth mentioning.
   Personally, and I know someone is going to blast me for this,
   I'm glad it was included in the language, even though I and
   nobody else ever uses it.  I always like having the choice to
   do things the way I like, which is truly what C programming is
   all about anyway.  The language doesn't dictate matters of
   style, that job is left to the programmer where it belongs.
   P.S. Unions also fall into the category of "why", but again,
   I think that just because a feature doesn't fit all situations,
   or even most situations, it is at least nice to know it's there
   for that one out of a million chance you might actually use it.

BUGS
   You shouldn't use this keyword enough to know if there are any
   bugs.  In fact, if you're still reading this, there's something
   wrong.  Here, directly below are a list of things you can use
   instead.

SEE ALSO
   for, while, do, switch, break, continue,
   THE WHOLE CONCEPT OF USING FUNCTIONS TO ENCAPSULATE CODE & DATA
```

1.17 /if

```
NAME
   if - Used to form (if-then) constructs.  Is the basic way to
        make decisions in C.

SYNOPSIS
   if (<conditional statement>)
   {
      <statement sequence>;
   }
   else if (<conditional statement>)
```

```
   {
      <statement sequence>;
   }
   else
   {
      <statement sequence>;
   }

FUNCTION
   If is used to form (if-then) constructs.  It is the basic way
   to make decisions in C.  Basically, anytime you have to choose
   between two courses of action based on some data or user input
   you use an if.  If tests for truth, (if such and such is true
   then do something or other else do this other thing).  Anything
   that is unequal to 0 is true, zero is false.  2+2 is true,
   2-2 is false...  If this makes little sense, don't worry, I'm
   not explaining it well anyway.  if you put a statement in your
   code like printf("%d\n", 2==2), then the output would most
   likely be 1 or -1, but could honestly be anything other than 0.
   Likewise printf("%d\n", 2==4), would output 0.  Usually when
   testing for truth you use the symbols > < >= <= != == && ||.
   >  greater than
   <  less than
   >= greater than or equal to
   <= less than or equal to
   != not equal to
   == equal to
   && and
   || or
   These are all logical operators as opposed to bitwise
   operators which are used to manipulate binary values.

INPUTS
   <conditional statement> - A statement which evaluates to either
                             true or false.  False is equal to 0
                             and true is anything else, however
                             it is common practice to have -1 or 1
                             represent true.
   <statement sequence> - The single statement or block of
                          statements which is/are to be executed.

RESULT
   none.

EXAMPLE
   if (x==3) printf("x is equal to 3\n");
   else
   {
      printf("OK, you were wrong.\n");
      printf("x is not equal to 3.\n");
   }

NOTES
   Get used to if.  You'll be using it a lot.

BUGS
   Aside from being iffy, "Ok, Ok, bad joke", none.
```

```
SEE ALSO
   switch.
```

1.18 /int

```
NAME
   int - Declares variables of type int, (quite possibly the most
         commonly used data type in C programming.)

SYNOPSIS
   int <variable list>;

FUNCTION
   int declares variables of type int.  Assuming that integers
   are a 16 bit data type, (on some computers they're 32 bits
   by default, but could be 64 or anything else depending on
   the hardware), an integer is any whole number which falls
   between the ranges of -32768 and 32767.  An integer must be
   at least that large.  A long integer must likewise be at
   least a 32 bit number meaning anything between -2147483648 and
   2147483647.

INPUTS
   <variable list> - Is a list of variables to be declared as
                     type int separated by commas.

RESULT
   none.

EXAMPLE
   int x, y = 2, z[5] = {1, 2, 3, 4, 5};

NOTES
   Integers are just about the fastest datatype in C.
   Calculations done with ints are lightning fast, plus
   by using the register keyword you can ask the C compiler
   to store them directly in the CPU's memory so access is
   even faster.  The down side of course is you can't do
   fractions in the traditional sense.  5/2 yields 2.  The
   remainder is simply chopped off.  To retain the whole
   answer with the remainder use the div function found in
   the stdlib.h.  There are ways you can get ints and long ints
   to do just about any job you want done, it just takes a little
   fore-thought.

BUGS
   Only works on whole numbers and can't hold very large numbers.

SEE ALSO
   void, char, float, double.

   long, short, signed, unsigned,
   const, volatile,
   extern, static, register, auto,
```

```
   struct, union, typedef, enum.
```

1.19 /long

```
NAME
   long - Is an access modifier, which can be used to preface
          either int or double to change their meaning, usually
          to increase their size.

SYNOPSIS
   long <data type> <variable list>;

FUNCTION
   Long is an access modifier, which can be used to preface either
   int or double to change their meaning, usually to increase
   their size.  long can be used by itself and it is understood
   that you mean long int.  A normal integer is usually 16 bits
   long, but a long integer is typically 32 bits long meaning
   instead of being able to only hold a max 32,767 a long can
   hold 2,147,483,647.  Similarly when used in reference to
   double it specifies that it should take up, (I think), 80
   bits instead of the normal 64.  This translates from something
   like 1.7 * 10^308 into 3.4 * 10^4932.  Don't quote me on
   that last one, I don't think long double's are very standard,
   and seriously, if you ever need a number that big, or that
   accurate, it probably means your a much better programmer than
   I, and you have no business reading this anyway unless of
   course it's for a good laugh.

INPUTS
   <data type> - In this case can only be int or double.

   <variable list> - A list of variables to be declared as
                     long, or long double.

RESULT
   none.

EXAMPLE
   long x, y = 2, z = 1000000;
                 // same as long int x, y = 2, z = 1000000;

NOTES
   Long is one of those funny things that got really screwed
   up when the ANSI folks decided to mess around with
   it's meaning.  Originally in the old Kernigan & Ritchie
   standard of C, long made a whole lot more sense.  You
   had the standard data types like char int and float.
   ints were either short or long, likewise floats too
   were either short or long.  If you didn't specify the
   keyword long then it was assumed to be short.  Ints were
   16 bits short 32 long, floats were 32 short and 64 long.
   In fact the keyword double and long float are synonymous.
   and long doubles didn't exist.  To me, that makes sense,
   but when the ANSI people came to standardize C compilers
```

```
   everywhere, (and they did a good job, I just don't like
   the way they did this), they not only said that we're now
   going to call long floats double and make a new long
   double type, but they made it illegal to make a long
   float.  If you try it, I'll lay you 10 to 1 odds your
   compiler will choke.  Anyway, so now years after the
   ANSI standard, long has a very obscure meaning now,
   at least in reference to what it meant originally.

BUGS
   Only works with int and double.  Otherwise none.

SEE ALSO
   void, int, char, float, double.

   short, signed, unsigned,
   const, volatile,
   extern, static, register, auto,
   struct, union, typedef, enum.
```

1.20 /register

```
NAME
   register - Is an access modifier which is used to specify
              to the compiler that the following variable
              declarations are such that speed is crucial
              and that the compiler should perform added
              optimizations on these variables.

SYNOPSIS
   register <data type> <variable list>;

FUNCTION
   Register is an access modifier which is used to specify to the
   compiler that the following variable declarations are such that
   speed is crucial and that the compiler should perform added
   optimizations on these variables.  Really this keyword should
   only be used on ints, which you would like to be stored in
   the CPU's registers so they take up less access time, (hence
   the name register).  Only ints can be stored in the CPU's
   registers, and it should be understood that this is only
   a request and not a command, the compiler may or may not
   store the variables in question in the CPU's registers,  it
   may or may not perform added optimizations on the variables
   that can't go into the registers.

INPUTS
   <data type> - Any C data type, (i.e. char, int, float, double)
   <variable list> - A list of variable being declared separated
                     by commas.

RESULT
   none.

EXAMPLE
```

```
   register int x, y = 0;

NOTES
   It's pretty much just a simple way to make you feel like
   you've optimized your code, when in fact the speed difference
   is minimal.  Really the best way to optimize a program is well
   thought out algorithms that do their job with as little pull
   on the CPU as possible.

BUGS
   It's only a request, it may or may not have any effect.

SEE ALSO
   void, char, int, float, double.

   long, short, signed, unsigned,
   const, volatile,
   extern, static, auto,
   struct, union, typedef, enum.
```

1.21 /return

```
NAME
   return - Exits a function immediately and a return value may
            be specified.

SYNOPSIS
   <data type> <function name>(<argument list>)
   {
      <statement sequence>;
      return <return value>;
   }

FUNCTION
   Return exits a function immediately and a return value may
   be specified.  Unless a function declares a void return type
   it must have at least one return statement that terminates
   the function and returns a value of the appropriate type.
   A function may have multiple return statements.

INPUTS
   <data type> - May be any of C's built in data types or it
                 may be a user defined (complex) data type, such
                 as a struct.
   <function name> - Any valid identifier which serves as the
                     name of the function.
   <argument list> - An argument list consists of variable
                     declarations separated by commas.  See
                     the example for clarification on this.
   <statement sequence> - The single statement or block of
                          statements which is/are to be executed.
   <return value> - The data to be passed back to the routine
                    that originally called the function.

RESULT
```

```
   What should I write for this, it is the result.  This is the
   mechanism by which a result is produced.

EXAMPLE
   int main(int argc, char *argv)
   {

      printf("Hello dare\n");
      return 0;
   }

NOTES
   none.
BUGS
   none.

SEE ALSO
```

1.22 /short

```
NAME
   short - Is an access modifier, which can be used to preface
           int to change it's meaning.

SYNOPSIS
   short int <variable list>;

FUNCTION
   short is an access modifier, which can be used to preface int
   to change it's meaning.  Actually, in practice, a short int
   is the same as saying int, and likewise short can be specified
   by itself where the definition short x; is the same as saying
   int x;  technically, ints can be 16 or 32 bits, (or really any
   number of bits depending on the machine), and short is any
   number of bits less than or equal to the default int.

INPUTS
   <variable list> - A list of variables to be declared separated
                     by commas.

RESULT
   none.

EXAMPLE
   short int x, y = 0;
      // same as short x, y = 0;

NOTES
   Short and long are two of a kind so you should take a look
   at what long is all about as well.  Short can only be applied
   to integers, while long can be applied to ints and doubles.
   Again I think the ANSI standard kind of changed the meaning of
   short, which is all right, and since I don't know how it was
   originally used, and don't have any pre-ANSI compilers to test
   it out on, I'm just going to leave a big hairy blank on the
```

```
   history.  In general short and long should only be used on
   ints, and on a more personal note, I've never used the short
   keyword once to my recollection.  Usually it's assumed that
   ints are shorts, and if your computer or compiler is more
   comfortable with 32 bit numbers than 16, why limit the size
   when there's no need.  Bottom line, you'll probably use long
   all the time, but probably forget short even exists.

BUGS
   none to my knowledge.

SEE ALSO
   long,

   void, int, char, float, double.

   signed, unsigned,
   const, volatile,
   extern, static, register, auto,
   struct, union, typedef, enum.
```

1.23 /signed

```
NAME
   signed - Is an access modifier which may be used on either
            character or integer data types.

SYNOPSIS
   signed <data type> <variable list>;

FUNCTION
   Signed is an access modifier which may be used on either
   character or integer data types.  Signed is used mainly
   to specify that a char can either be negative or positive.
   Doing this causes the maximum number a char can hold to be
   cut in half.  Really all variable types, including char, are
   signed by default so I'm not sure that there is ever an
   instance where you would want to use this modifier.

INPUTS
   <data type> - In this case either char or int.
   <variable list> - A list of variables to be declared separated
                     by commas.

RESULT
   none.

EXAMPLE
   signed char x, y = -100;
   printf("%d\n", y);

NOTES
   Notice that in the example above when I printed y with
   printf I used a %d instead of %c which is normally used
   for characters.  The reason is, I'm using the type char
```

```
   here to store numbers and not characters in the normal sense.

BUGS
   none.

SEE ALSO
   unsigned,

   void, int, char, float, double.

   long, short, const, volatile,
   extern, static, register, auto,
   struct, union, typedef, enum.
```

1.24 /sizeof

```
NAME
  sizeof - Is used to calculate the size of variables and other
           objects.

SYNOPSIS
  size_t sizeof(<variable type>)<expression>;
  or
  size_t sizeof <variable> <expression>;

FUNCTION
   Sizeof is used to calculate the size of variables and other
   objects.  Sizeof is useful in determining the size of a
   variable or other object, but it's most useful application is
   in calculating the size for an object that is being dynamically
   allocated.  Basically malloc() and sizeof go together like
   peanut butter and jelly.

INPUTS
   <variable type> - Can be any of C's built in type's or a user
                     defined type like a struct or something.
   <variable> - Or you can use the variable itself.  sizeof will
                return how much memory it's taking up.
   <expression> - Any expression which you might want to use to
                  modify the number that sizeof returns.

RESULT
   size_t - Is the size of the object passed to sizeof in bytes.
            size_t is defined in stddef.h.  Technically speaking,
            sizeof figures an objects size as a multiple of the
            char type, (which is almost always a byte).  If your
            computer stores characters in something larger than
            an 8 bit byte, then you might have to rethink your
            implementation.  In practice though, I've never heard
            of this happening.

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

```
   struct test
   {
      int i;
      char c;
   } teststruct; typedef struct test test;

   int main(void)
   {
      int x, *ptr;
      ptr = (int *)malloc(sizeof(int)*5);
      for (x = 0; x < 5; x++)
      {
          *(ptr+x) = x;
          printf("%d\n", *(ptr+x));
      }
      printf("Won't work, sizeof ptr isn't %d\n", sizeof(ptr));
      printf("size of x is %d\n", sizeof(x));
      printf("size of teststruct is %d\n", sizeof(teststruct));
      printf("testtruct should be sizeof(int)+sizeof(char).\n");
      printf("just to test, sizeof(int)+sizeof(char)==%d\n",
         sizeof(int)+sizeof(char));
      return 0;
   }

NOTES
   Sizeof is a unary operator.  I almost always use it as though
   it were a function, but it is an operator none the less.
   malloc and it's related functions calloc realloc and free
   which are used to allocate memory from the operating system
   are where sizeof gets used most often.
   Also the type size_t is defined in stddef.h.

BUGS
   none.

SEE ALSO
   size_t, malloc, calloc, realloc, free.
```

1.25 /static

```
NAME
   static - Is used to specify that a variable is to remain in
            memory even when the function or block to which
            it is assigned goes out of scope.

SYNOPSIS
   static <data type> <variable list>

FUNCTION
   Static is used to specify that a variable is to remain in
   memory even when the function or block to which it is assigned
   goes out of scope.  Simply stated, normally when you exit a
   function, all local variables in that function are destroyed
   so the memory can be used elsewhere, (i.e., when you go back
   into that function later, the variables are not the same as
```

when you left the function.) The static keyword specifies that you don't want the memory to be dumped, so when you come back into the function, everything is just as you left it. There are two practical uses for static variables that come blaringly to mind. One is when you need to keep track of some bit of info between function calls, (like maybe how many times you've called the function or something). The other is when you want to return a pointer to info generated by the function, but don't want to declare a global variable and call by value isn't economical.

There is a another and somewhat confusing use of the keyword static, which is to specify that a global identifier, (which can be any identifier including a variable or function), is to have internal linkage. What this means is basically that say you have two source code files and both of them contain either functions and/or variables with the same names. If you try to compile them together, you'll get an error from the compiler. If you declare the vars and or functions in question with the static keyword it tells the compiler that the source code is useing the variable within itself and not the one that was defined in the other source code file. At first this looks like a useless feature that no-one in their right mind would use, but it does have one VERY important use. The problem with global variables is that they can very quickly polute your code which is why you're encouraged to use static variables within a function. Sometimes however, you have to use a global var to get the job done, and there's no way around it. Well if you're working on a large project made up of multiple files you can use the static keyword when you define your global variable, and it will in effect shield all the other source file segments from your global variable. They won't be aware that it even exists. I suggest you play with this a little, as it's an extremely useful tool for allowing you to "cheat" the system, and still have super clean code.

INPUTS
<data type> - Any of C's built in data types, or a user defined (complex) data type of your own.
<variable list> - A list of variables being declared separated by commas.

RESULT
none.

EXAMPLE

```
#include <stdio.h>

int *test(int a, int b);

int main(void)
{
   int x, *mainptr;

   mainptr = test(0, 10);
   for (x = 0; x < 10; x++) printf("%d\n", *(ptr+x));
   printf("\n\n\n");
```

```
      mainptr = test(5, 15);
      for (x = 0; x < 15; x++) printf("%d\n", *(ptr+x));

      return 0;
   }

   int *test(int a, int b)
   {
      int x;
      static int *ptr = NULL;

      if (ptr==NULL) ptr = (int *)malloc(sizeof(int)*b);
      else ptr = (int *)realloc(ptr, sizeof(int)*b);
      for (x = a; x < b; x++) *(ptr+x) = x;
      return ptr;
   }

NOTES
   Static shouldn't be used when it's not needed, but it is an
   invaluable tool for maintaining the structure of a program
   when trying to do some things.  Mainly static's main purpose
   for existing is to limit the need for global variables.
   Indeed even when the use of global variables is necessary,
   the static keyword can be used to limit the impact they have
   on the overall program.

BUGS
   none.  Sorry if I didn't explain this keyword well.  It's
   a little hard for me to word.  When all else fails,
   experiment, a little old fashioned hacking never hurt
   anyone.

SEE ALSO
   auto, extern, register, const, volatile,

   void, char, int, float, double.
   long, short, signed, unsigned,
   struct, union, typedef, enum.
```

1.26 /struct

```
NAME
   struct - Is used to create user defined (complex) variable
            types on top of C's built in types.

SYNOPSIS
   struct <type name>
   {
      <variable declarations>;
   } <global variable list>;

FUNCTION
   Struct is used to create user defined (complex) variable
   types on top of C's built in types.  Basically struct allows
   you to structure a bunch of variables into a logical grouping.
```

```
   If your familiar with the concept and terminology of databases
   a struct is like making a field, which will later contain
   a whole bunch of data pertaining to a single subject.
   If the idea is unclear, try looking at the example for
   clarification.

INPUTS
   <type name> - Is the name of the new type you are creating.
                 Think of this as roughly corresponding to int,
                 char, or float or the like with the exception
                 that this is your own custom variable type.
   <variable declarations> - Several lists of variable
                             declarations which may be made up of
                             any of C's built in types, or of
                             other user defined (complex) types
                             defined elsewhere.
   <global variable list> - A list of variables of this newly
                            defined type separated by commas.  Any
                            variables declared here are considered
                            global.  You may declare global
                            variables of this type elsewhere but
                            it is considered proper to do it here.

RESULT
   none.

EXAMPLE
   #include <stdio.h>
   #include <string.h>

   struct phonepage
   {
      char name[257];
      int areacode, localcode, fourdigitcode;
      char address[257];
   } phonebook[10];

   // the following line is just so you can type phonepage
   // instead of typing struct phonepage to declare a variable
   // of that type.  THIS IS NOT A NECESSARY STATEMENT.
   // It is intended to make the meaning of the statements in main
   // stand out more clearly.
   typedef struct phonepage phonepage;

   int main(void)
   {
      phonepage myfriend;

      strcpy(myfriend.name, "Harry");
      myfriend.areacode = 900;
      myfriend.localcode = 555;
      myfriend.fourdigitcode = 1234;
      strcpy(myfriend.address,
         "Beverly Hill 90210, <Yeah right ;-)>");

      phonebook[0] = myfriend;
```

```
      printf("My friend's name is %s\n", phonebook[0].name);
      printf("My friend's phone number is (%d) %d-%d\n",
         phonebook[0].areacode, phonebook[0].localcode,
         phonebook[0].fourdigitcode);
      printf("my friend's address is %s\n", phonebook[0].address);

      return 0;
   }

NOTES
   When I first started learning C, the syntax of how to construct
   a struct really baffled me, mostly because in many of
   the books I was reading each programmer had a different way
   of doing it.  Specifically the difference between the name at
   the beginning and the list of names that follows the definition
   threw me for a loop.  It was never impressed on me by the
   different authors, that the name at the top is a new variable
   type, and the list of names at the end are simply variables of
   that type.  Structs are really the first glimmer of the object
   oriented way of looking at a problem.  C is not an object
   oriented language, but structs promote the grouping of data
   into logical/conceptual objects.  C++ takes the next step and
   lets you put code into your own variables as well as data.
   There's more to object oriented programming then just that,
   but I'm just trying to impress that if you've no idea what
   object oriented programming is, Structs definitely fall into
   the spirit of object oriented programming.
   P.S. a structure can also be used to create a bit field.
   A bit field is useful when you want to pack as much info
   into a given piece of memory as is humanly possible.
   This is an advanced technique which I'm too lazy to go into
   in depth here.  If you think you might need one, or are just
   interested, or need the syntax, pick up any book on C
   programming and look in the index for bit fields.
   A bit field looks something like this,
   struct test
   {
      int bool1 : 1;
      int bool2 : 1;
      int byte1 : 8;
      int byte2 : 8;
   } ;

BUGS
   none.

SEE ALSO
   typedef, union, enum,

   void, char, int, float, double.
   long, short, signed, unsigned,
   const, volatile,
   extern, static, register, auto.
```

1.27 /switch

```
NAME
   switch - A statement used to choose between alternate courses
            of action based on an integer or enumeration value.

SYNOPSIS
   switch (<variable>)
   {
      case <const value>:
         <statement sequence>;
         break; // if this is omitted, 2nd case will execute too.
      case <const value>:
         <statement sequence>;
         break;
      default:
         <statement sequence>;
   }

FUNCTION
   Switch is a statement used to choose between alternate courses
   of action based on an integer or enumeration value.  Switch
   is similar in purpose to an If-Then, but is more restrictive.
   Essentially all a switch can test for is equality.  When a
   single variable must be tested multiple times for equality
   and a simple inequality won't suffice, a switch statement
   can add more clarity to the code then an if-else if ladder.

INPUTS
   <variable> - The variable that is to be tested in the body of
                the switch statement.
   <const value> - Any constant expression that evaluates to an
                   integer number.  If the <variable> is equal
                   to this number, then the statement sequence
                   directly after is executed.  const value may
                   not contain any variables.
   <statement sequence> - The single statement or block of
                          statements which is/are to be executed.

RESULT
   none.

EXAMPLE
   #include <stdio.h>

   int main(void)
   {
      int x = 2;

      switch (x)
      {
         case 1:
            printf("1\n");
            break;
         case 2:
            printf("2\n");
```

```
            break;
         default:
            printf("3\n");
      }
      return 0;
   }

NOTES
   Switch is often used when the user is supposed to select
   one of several options.  Menu programs, and similar progs
   that prompt for user input can benefit from the use of
   switches.  Also, remember, switch only works with integral
   types.  This could be a char, int or enumeration.

BUGS
   none.
SEE ALSO
   case, break, default, int, enum,
   if, else.
```

1.28 /typedef

```
NAME
   typedef - Is a C keyword for creating an alias for data types.

SYNOPSIS
   typedef <old name> <new name>;

FUNCTION
   Typedef is a C keyword for creating an alias for data types.
   Typedef can be used to substitute your own names for variable
   types, like typedef unsigned char byte;.  Also you can use it
   to abbreviate long type definitions so their not such a pain to
   type.

INPUTS
   <old name> - This is the old data type name.  It can be made up
                of several names, like unsigned char or
                struct test.
   <new name> - This is the alias of the data type.  This
                can't contain any spaces as C assumes the last
                word in a typedef statement is the alias.  Also
                it should be noted that you may specify that the
                new type should inherently an array or pointer or
                something, (i.e., typedef char strtype[257]; or
                typedef int *intptr;).

RESULT
   none.

EXAMPLE
   typedef char strtype[257];
   typedef int *intptr;
   typedef struct
   {
```

```
      strtype str;
      int x, y, z;
   } structtype;

NOTES
   The last part of the example is a little ambiguous and deserves
   some explanation.  In a normal struct definition, any name that
   follows the closing bracket is the name of a global variable
   that you are declaring.  In this instance typedef is saying to
   replace structtype with the whole struct definition.  To
   demonstrate what I'm talking about, try compiling the following
   example.
   #include <stdio.h>
   #include <string.h>

   int main(void)
   {
      struct {int x, y, z; char str[257];} hello;

      strcpy(hello.str, "Hello There\n");

      printf("%s\n", hello.str);

      return 0;
   }
   In this code fragment, hello is a variable, and the part that
   says struct {int x, y, z; char str[257];} IS the data type.
   you are basically assigning the variable hello a unique data
   type that can't be assigned anywhere else unless of course you
   were to re-specify the struct from scratch.

BUGS
   none.
SEE ALSO
   struct, union, enum, void, int, char, float, double.
```

1.29 /union

```
NAME
   union - Similar to struct, but specifies that it's data members
           are to occupy the same area of memory.

SYNOPSIS
   union <type name>
   {
      <variable declarations>;
   } <global variable list>;

FUNCTION
   Union is similar to struct, but specifies that it's data
   members are to occupy the same area of memory.  So, if you
   specified a union (union test {int x; char c;} ;) then
   any data assigned to c is also assigned to the low order
   bits of x, and any value assigned to x can also be accessed
   by c assuming that value is no larger than 8 bits in which
```

```
   case the value returned by c is truncated.

INPUTS
   <type name> - Is the name of the new type you are creating.
                 Think of this as roughly corresponding to int,
                 char, or float or the like with the exception
                 that this is your own custom variable type.
   <variable declarations> - Several lists of variable
                             declarations which may be made up of
                             any of C's built in types, or of
                             other user defined (complex) types
                             defined elsewhere.
   <global variable list> - A list of variables of this newly
                            defined type separated by commas.  Any
                            variables declared here are considered
                            global.  You may declare global
                            variables of this type elsewhere but
                            it is considered proper to do it here.

RESULT
   none.

EXAMPLE
   #include <stdio.h>
   #include <values.h>

   union test
   {
      int x;
      char c;
   } ;

   int main(void)
   {
      union test aunion;

      aunion.x = MAXINT;

      printf("%d\n", aunion.x);

      aunion.c = 'a';

      printf("%d\n", (int)aunion.c);
      printf("%d\n", aunion.x);

      return 0;
   }


NOTES
   Unions only have a few very specialized applications in low
   level programming.  For instance when writing the malloc
   function, programmers generally use a union to control how the
   memory is allocated.  For application programmers, (That's you
   and me), I can't off hand think of any situation that would
   warrant the use of a union.
```

```
BUGS
   none.

SEE ALSO
   struct, enum, typedef, void, int, char, float, double.
```

1.30 /unsigned

```
NAME
   unsigned - Is an access modifier which may be used on either
              character or integer data types.

SYNOPSIS
   unsigned <data type> <variable list>;

FUNCTION
   Unsigned is an access modifier which may be used on either
   character or integer data types.  Unsigned is used mainly
   to extend the maximum value an integer, (or other integral
   type), may hold.  (An integral type is any built in,
   non-floating point, type.  Definitely not complex types.)

INPUTS
   <data type> - In this case either char or int.
   <variable list> - A list of variables to be declared separated
                     by commas.

RESULT
   none.

EXAMPLE
   unsigned int x, y = 40000;
   printf("%d\n", y);

NOTES
   Normally an integer value is 16 bits long, translating into
   any number between -32767 and 32767.  Making an int unsigned
   will effectively change the range to 0 through 65,??? basically
   doubling highest possible positive integer.  This can be
   useful when you need numbers slightly higher than 32767 but
   don't need to store any negative values, like in a loop
   counter or something.

BUGS
   none.

SEE ALSO
   signed,

   void, int, char, float, double.

   long, short, const, volatile,
   extern, static, register, auto,
   struct, union, typedef, enum.
```

1.31 /void

```
NAME
   void - Used to specify that a function doesn't return any
          return values, or can declare void pointers which
          can hold a pointer of any type.

SYNOPSIS
   void <function name>();
      or
   void *<var or func name>;

FUNCTION
   Void is used to specify that a function doesn't return any
   return values, or can declare void pointers which
   may be cast into any other ptr type without explicit casting.
   note that C++ is a little more strict about using a void
   ptr as an implicit cast and you may have to go ahead and
   cast it explicitly if your using a C++ compiler.

INPUTS
   <function name> - Any valid function name which will have a
                     void return type specifying that this
                     function has no return type.
   <var or func name> - If it is a variable name then you
                        are specifying a void ptr variable type.
                        if it is a function name then you
                        are specifying a void ptr return type.
                        A void ptr return type does not indicate
                        no return value.  It specifies that the
                        type of the pointer which is being
                        returned is generic and may be any type.

RESULT
   none.

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

   int main(void)
   {
      int *x;
      x = malloc(sizeof(int));

      *x = 67;

      printf("%d\n", *x);

      return 0;
   }

NOTES
   In the example above, malloc returns a void ptr.  This
   allows x to be assigned the value even though it is an
   integer pointer.
```

```
   In addition to all I've said so far, I think that void used as
   a return type signifying no return type, was instituted by the
   ANSI standard.  I believe they did it this way because of
   some incompatibility with the previous syntax for defining
   C functions in the Ritchie/Kernigan definition.  I think
   they wanted to maintain compatibility with programs using the
   older definitions, but needed to find a way to distinguish
   between the two.  I don't know the details as I'm not very
   familiar with that definition of C since it is quit old,
   (at least a decade now).  A lot of old code uses it though
   so it's useful to know the distinction.  You should use the
   new syntax though as it allows better type checking by the
   compiler, and on top of that I recommend always declaring
   return types even if the return type is int, because the
   latest definition of C++ doesn't support implicit int
   return types anymore.  "If your like me, you use C AND
   C++ compilers to compile your C programs, (depending on the
   tools you have available to you at the time), plus it's just
   a good idea to make your C programs compatible with C++
   just in case you ever want to go to it later.  You might
   want to mix your old routines with your new code or something.

BUGS
   none.

SEE ALSO
   signed,

   void, int, char, float, double.

   long, short, const, volatile,
   extern, static, register, auto,
   struct, union, typedef, enum.
```

1.32 /volatile

```
NAME
   volatile - Is an access modifier which specifies that a given
              variable may be altered by another program outside
              the current program's control and for the compiler
              to perform no optimizations that would cause an
              error should such an outside alteration happen.

SYNOPSIS
   volatile <data type> <variable list>;

FUNCTION
   Volatile is an access modifier which specifies that a given
   variable may be altered by another program outside the current
   program's control and for the compiler to perform no
   optimizations that would cause an error should such an outside
   alteration happen.  Most frequently, this type of variable is
   used when you want your program to access some fixed hardware
   (or software) resource which is regulated by the OS and/or
```

```
   other programs must access and/or change it.  A good example
   of this is if you wanted to access the computer's clock
   directly.  You'd assign a pointer to wherever the time info
   is kept in the computer, and whenever you'd want to know
   the time you could just take a look at that pointer.

INPUTS
   <data type> - Any of C's built in types or your own user
                 defined (complex) data type.
   <variable list> - A list of variables being declared separated
                     by commas.

RESULT
   none.

EXAMPLE
   volatile int x;

NOTES
   Anytime one uses the volatile keyword, it denotes that your
   getting pretty close to the system, IE pretty low level.  If
   your trying to write portable software that will compile on
   multiple platforms, (which should be the goal of every good
   little programmer), it's not a good idea to start accessing
   hardware, which may or may not exist on other platforms or
   even later models of the same type of computer, (DOES AGA
   RING ANY BELLS HERE).  In closing, this keyword is best left
   to system and compiler programmers.

   PS, Don't take my word on this as gospel, I'm not very
   familiar with volatile and it's uses.

BUGS
   none.

SEE ALSO
   const, extern, static, register, auto.
```

1.33 /while

```
NAME
   while - Used to form a while loop, the most basic looping
           structure in C.

SYNOPSIS
   while (<condition>)
   {
      <statement sequence>;
   }

FUNCTION
  While is used to form a while loop, the most basic looping
  structure in C.  While executes while the <condition> is true,
  when the condition becomes false the loop exits.  The test
  condition is evaluated first when you enter the loop and then
```

```
  once every time the loop finishes a cycle.

INPUTS
   <condition> - Any valid C expression that evaluates to either
                 true of false.
   <statement sequence> - The single statement or block of
                          statements which is/are to be executed.

RESULT
   none.

EXAMPLE
   x = 0;
   while (x<10)
   {
      x++;
      printf("%d\n", x);
   }

NOTES
   While and do while differ only in that a do while evalutes it's
   conditional statement after the first iteration and while
   evaluates it's expression before the loop is executed.  This
   distinction can have a profound effect on what happens if
   one is unaware of the distinction, however the loops can
   generally substitute each other with minor modification.

BUGS
   none.

SEE ALSO
   do, for, switch, break, continue, default, if, else.
```