# Introduction to C Programming

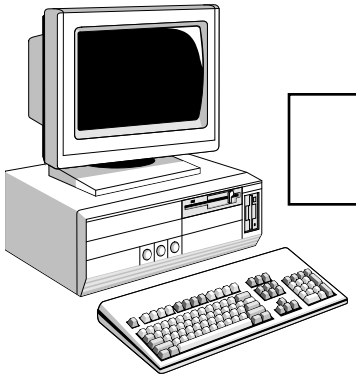
# A Brief History

- Created by Dennis Ritchie at AT&T Labs in 1972
- Originally created to design and support the Unix operating system.
- There are only 27 keywords in the original version of C.
  - for, goto, if, else ……
- Easy to build a compiler for C.
  - Many people have written C compilers
  - C compilers are available for virtually every platform
- In 1983 the American National Standards Institute (ANSI) formed a committee to establish a standard definition.
  - Called ANSI Standard C.
  - As opposed to K&R C (referring to the general “standards” that appeared in the first edition of Brian Kernighan and Ritchie’s influential book: *The C Programming Language*)

# Why use C?

- C is intended as a language for programmers
  - BASIC was for nonprogrammers to program and solve simple problems.
  - C was created, influenced, and field-tested by working programmers.
- C is powerful and efficient
  - You can nearly achieve the efficiency of assembly code.
  - System calls and pointers allow you do most of the things that you can do with an assembly language.
- C is a structured language
  - Code can be written and read much easier.
- C is standardized
  - Your ANSI C program should work with any ANSI C compiler.

# The C Development Cycle



Edit Program

Source Code

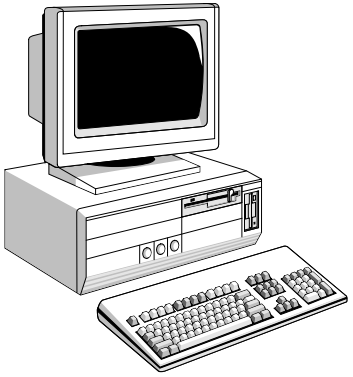
Compile

Object Code

Library Files

Link Object Code

Executable



# “Hello World”

- Everyone writes this program first

```
#include <stdio.h>
int main ( )
{
    printf ("Hello, World!\n");
    return 0;
}
```

# Compilation (1)

- Compilation translates your source code (in the file hello.c) into object code (machine dependent instructions for the particular machine you are on).
  - Note the difference with Java:
    - The javac compiler creates Java byte code from your Java program.
    - The byte code is then executed by a Java virtual machine, so it's machine independent.
- Linking the object code will generate an executable file.
- There are many compilers for C under Unix
  - SUN provides the Workshop C Compiler, which you run with the cc command
  - There is also the freeware GNU compiler gcc

# Compilation (2)

- To compile a program:
  - Compile the program to object code.

    obelix[2] > cc –c hello.c
  - Link the object code to executable file.

    obelix[3] > cc hello.o –o hello
- You can do the two steps together by running:

  obelix[4] > cc hello.c –o hello
- To run your program:

  obelix[5] > ./hello

  Hello World!

If you leave off the -o, executable goes into the file a.out

# Compilation (3)

- Error messages are a little different than you may be used to but they can be quite descriptive.
- Suppose you forgot the semi-colon after the printf

```
obelix[3] > cc hello.c –o hello
"hello.c", line 5: syntax error before or at: return
cc: acomp failed for hello.c
```

- Notice that the compiler flags and informs you about the error at the first inappropriate token.
  - In this case, the return statement.
- Always try to fix problems starting with the first error the compiler gives you - the others may disappear too!

# Example 1

```
#include <stdio.h>
int main ()
{
    int radius, area;

    printf ("Enter radius (i.e. 10) : ");
    scanf ( "%d", &radius);
    area = 3.14159 * radius * radius;
    printf ("\nArea = %d\n\n", area);
    return 0;
}
```

# Example 2

```
#include <stdio.h>
int main ()
{
    int i, j;
    for (i = 0; i < 10; i++)
    {
            printf ("\n");
            for (j = 0; j < i+1; j++ )
                printf ( "A");
    }
    printf("\n");
    return 0;
}
```

# Example 3

```
/* Program to calculate the product of
     two numbers */

#include <stdio.h>

int product(int x, int y);

int main ()

{

   int a,b,c;

   /* Input the first number */

   printf ("Enter a number between 1
              and 100: ");

   scanf  ("%d", &a);


   /* Input the second number */

   printf ("Enter another number
              between 1 and 100: ");

   scanf  ("%d", &b);

   /* Calculate and display the product */

   c = product (a, b);

   printf ("%d times %d = %d \n", a, b, c);

   return 0;

}


/* Functions returns the product of its
     two arguments */

int product (int x, int y)

{

   return (x*y);

}
```

# Basic Types and Formatted I/O

# C Variables Names (1)

Variable Names

- Names may contain letters, digits and underscores
- The first character must be a letter or an underscore.
  - the underscore can be used but watch out!!
- Case matters!
- C keywords cannot be be used as variable names.

```
present, hello, y2x3, r2d3, ...     /* OK */
_1993_tar_return                    /* OK but don't */
Hello#there                         /* illegal */
double                              /* shouldn't work */
2fartogo                            /* illegal */
```

# C Variables Names (2)

Suggestions regarding variable names

- DO: use variable names that are descriptive
- DO: adopt and stick to a standard naming convention
  - sometimes it is useful to do this consistently for the entire software development site

- AVOID: variable names starting with an underscore
  - often used by the operating system and easy to miss
- AVOID: using uppercase only variable names
  - generally these are pre-processor macros (later)

# C Basic Types (1)

- There are only a few basic data types in C
  - char: a single byte, capable of holding one character
  - int: an integer of fixed length, typically reflecting the natural size of integers on the host machine (i.e., 32 or 64 bits)
  - float: single-precision floating point
  - double: double precision floating point

# C Basic Types (2)

- There are a number of qualifiers which can be applied to the basic types
  - length of data
    - short int:
      - "shorter" int, <= number of bits in an int
      - can also just write "short"
    - long int:
      - a "longer int", >= number of bits in an int
      - often the same number of bits as an int
      - can also just write "long"
    - long double
      - generally extended precision floating point
  - signed and unsigned
    - unsigned int
      - an int type with no sign
      - if int has 32-bits, range from 0..$2^{32}$-1
      - also works with long and short
    - unsigned char
      - a number from 0 to 255
    - signed char
      - a number from –128 to 127 (8-bit signed value)
      - very similar to byte in Java

# C Basic Types (3)

- All types have a fixed size associated with them
  - this size can be determined at compile time
- Example storage requirements

| DATA | Bytes Required |
|---|---|
| The letter *x* (char) | 1 |
| The number 100 (int) | 4 |
| The number 120.145 (double) | 8 |

- These numbers are highly variable between C compilers and computer architectures.
- Programs that rely on these figures must be very careful to make their code portable (ie. Try not to avoid relying on the size of the predefined types)

# C Basic Types (4)

Numeric Variable Types

- Integer Types:
  - Generally 32-bits or 64-bits in length
  - Suppose an int has b-bits
    - a signed int is in range $-2^{b-1}..2^{b-1}-1$
      - -32768 .. 32767 (32767+1=-32768)
    - an unsigned int is in range $0..2^{b}-1$
      - 0 .. 65535 (65535+1=0)
    - no error message is given on this "overflow"
- Floating-point Types:
  - Generally IEEE 754 floating point numbers
    - float (IEEE single): 8 bits exponent, 1-bit sign, 23 bits mantissa
    - double (IEEE double): 10 bits exponent, 1-bit sign, 53 bits mantissa
    - long double (IEEE extended)
  - Only use floating point types when really required
    - they do a lot of rounding which must be understood well
    - floating point operations tend to cost more than integer operations

# C Basic Types (5)

A typical 32-bit machine

| Type | Keyword | Bytes | Range |
|---|---|---|---|
| character | char | 1 | -128...127 |
| integer | int | 4 | -2,147,483,648...2,147,438,647 |
| short integer | short | 2 | -32768...32367 |
| long integer | long | 4 | -2,147,483,648...2,147,438,647 |
| long long integer | long long | 8 | -9223372036854775808 … 9223372036854775807 |
| unsigned character | unsigned char | 1 | 0...255 |
| unsigned integer | unsigned int | 2 | 0...4,294,967,295 |
| unsigned short integer | unsigned short | 2 | 0...65535 |
| unsigned long integer | unsigned long | 4 | 0...4,294,967,295 |
| single-precision | float | 4 | 1.2E-38...3.4E38 |
| double-precision | double | 8 | 2.2E-308...1.8E308 |

# Formatted Printing with printf (1)

- The printf function is used to output information (both data from variables and text) to standard output.
  - A C library function in the <stdio.h> library.
  - Takes a format string and parameters for output.
- printf(format string, arg1, arg2, …);
  - e.g. printf("The result is %d and %d\n", a, b);
- The format string contains:
  - Literal text:  is printed as is without variation
  - Escaped sequences:  special characters preceeded by \
  - Conversion specifiers: % followed by a single character
    - Indicates (usually) that a variable is to be printed at this location in the output stream.
    - The variables to be printed must appear in the parameters to printf following the format string, in the order that they appear in the format string.

# Formatted Printing with printf (2)

- Conversion Specifiers

| Specifier | Meaning |
|---|---|
| %c | Single character |
| %d | Signed decimal integer |
| %x | Hexadecimal number |
| %f | Decimal floating point number |
| %e | Floating point in “scientific notation” |
| %s | Character string (more on this later) |
| %u | Unsigned decimal integer |
| %% | Just print a % sign |
| %ld, %lld | long, and long long |

- There must be one conversion specifier for each argument being printed out.
- Ensure you use the correct specifier for the type of data you are printing.

# Formatted Printing with printf (3)

- Escape Sequences:

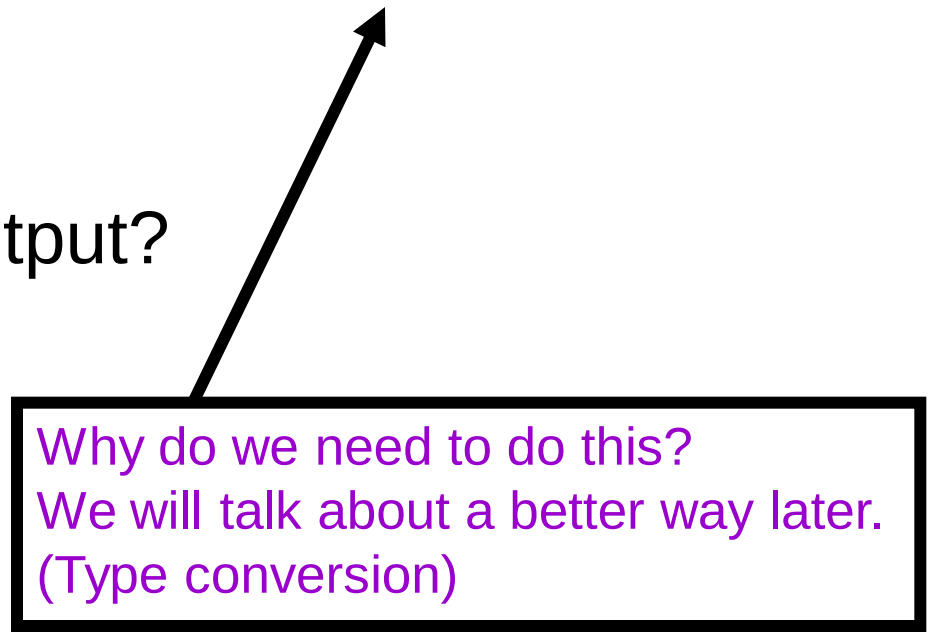
| Sequence | Meaning |
|---|---|
| \a | Bell (alert) |
| \b | Backspace |
| \n | Newline |
| \t | Horizontal tab |
| \\ | Backslash |
| \' | Single quote |
| \" | Double quotation |
| \x*hh* | ASCII char specified by hex digits *hh* |
| *ooo* | ASCII char specified by octal digits *ooo* |

# Formatted Printing with printf (4)

- An example use of printf

```
#include <stdio.h>
int main() {
   int ten=10,x=42;
   char ch1='o', ch2='f';
   printf("%d%% %c%c %d is %f\n",
          ten,ch1,ch2,x, 1.0*x / ten );
   return 0;
}
```

- What is the output?



# Reading Numeric Data with scanf (1)

- The scanf function is the input equivalent of printf
  - A C library function in the <stdio.h> library
  - Takes a format string and parameters, much like printf
  - The format string specifiers are nearly the same as those used in printf
- Examples:

```
scanf ("%d", &x);       /* reads a decimal integer */
scanf ("%f", &rate);    /* reads a floating point value */
```

- The ampersand (&) is used to get the “address” of the variable
  - All the C function parameters are “passed by value”.
  - If we used scanf("%d",x) instead, the value of x is passed. As a result, scanf will not know where to put the number it reads.
  - More about this in Section 15 (“Functions”)

# Reading Numeric Data with scanf (2)

- Reading more than one variable at a time:
  - For example:

    ```
    int n1, n2;  float f;
    scanf("%d%d%f",&n1,&n2,&f);
    ```

  - Use white spaces to separate numbers when input.

    ```
    5      10      20.3
    ```

- In the format string:
  - You can use other characters to separate the numbers

    ```
    scanf("value=%d,ratio=%f", &value,&ratio);
    ```

    - You must provide input like:

      ```
      value=27,ratio=0.8
      ```

  - scanf returns an int
    - If end-of-file was reached, it returns EOF, a constant defined in <stdio.h>
    - Otherwise, it returns the number of input values correctly read from standard input.

# Reading Numeric Data with scanf (3)

- One tricky point:
  - If you are reading into a long or a double, you must precede the conversion specifier with an `l` (a lower case L)
  - Example:

```
int main() {
   int x;
   long y;
   float a;
   double b;
   scanf("%d %ld %f %lf", &x, &y, &a, &b);
   return 0;
}
```

# Type Conversion

- C allows for conversions between the basic types, implicitly or explicitly.
- Explicit conversion uses the cast operator.
- Example 1:

```
int x=10;
float y,z=3.14;
y=(float) x;       /* y=10.0 */
x=(int) z;         /* x=3     */
x=(int) (-z);      /* x=-3   -- rounded approaching zero */
```

- Example 2:

```
int i;
short int j=1000;
i=j*j;                /* wrong!!! */
i=(int)j * (int)j;   /* correct */
```

# Implicit Conversion

- If the compiler expects one type at a position, but another type is provided, then implicit conversion occurs.
- Conversion during assignments:

```
char c='a';
int i;
i=c;    /* i  is assigned the ASCII code of ‘a’ */
```

- Arithmetic conversion – if two operands of a binary operator are not the same type, implicit conversion occurs:

```
int i=5 , j=1;
float x=1.0 , y;
y = x / i;          /* y = 1.0 /  5.0  */
y = j / i;          /* y = 1 / 5  so  y = 0 */
y = (float) j / i;   /* y = 1.0 / 5  */
                    /* The cast operator has a higher precedence */
```

# Example

The sizeof() function returns the number of bytes in a data type.

```
int main() {
  printf("Size of char ..... = %2d byte(s)\n", sizeof(char));
  printf("Size of short ..... = %2d byte(s)\n", sizeof(short));
  printf("Size of int ..... = %2d byte(s)\n", sizeof(int));
  printf("Size of long long …… = %2d byte(s)\n", sizeof(long long));
  printf("Size of long ..... = %2d byte(s)\n", sizeof(long));
  printf("Size of unsigned char. = %2d byte(s)\n", sizeof (unsigned char));
  printf("Size of unsigned int.. = %2d byte(s)\n", sizeof (unsigned int));
  printf("Size of unsigned short = %2d byte(s)\n", sizeof (unsigned short));
  printf("Size of unsigned long. = %2d byte(s)\n", sizeof (unsigned long));
  printf("Size of float ..... = %2d byte(s)\n", sizeof(float));
  printf("Size of double ..... = %2d byte(s)\n", sizeof(double));
  printf("Size of long double .. = %2d byte(s)\n", sizeof(long double));
  return 0;
}
```

# Example

Results of a previous run of this code on obelix ...

| | |
|---|---|
| Size of char | = 1 byte(s) |
| Size of short | = 2 byte(s) |
| Size of int | = 4 byte(s) |
| Size of long | = 4 byte(s) |
| Size of long long | = 8 byte(s) |
| Size of unsigned char | = 1 byte(s) |
| Size of unsigned int | = 4 byte(s) |
| Size of unsigned short | = 2 byte(s) |
| Size of unsigned long | = 4 byte(s) |
| Size of float | = 4 byte(s) |
| Size of double | = 8 byte(s) |
| Size of long double | =16 byte(s) |

# Creating Simple Types

- typedef creates a new name for an existing type
  - Allows you to create a new name for a complex old name
- Generic syntax

```
typedef  oldtype  newtype;
```

- Examples:

```
typedef  long  int32;    /* suppose we know an int has 32-bits */
typedef  unsigned char  byte;  /* create a byte type */
typedef  long double extended;
```

- These are often used with complex data types
  - Simplifies syntax!

# Variable Declaration (1)

- Generic Form

  typename varname1, varname2, ...;

- Examples:

```
int  count;
float a;
double percent, total;
unsigned char x,y,z;
long int aLongInt;
long AnotherLongInt
unsigned long a_1, a_2, a_3;
unsigned long int b_1, b_2, b_3;
typedef long int32;
int32 n;
```

- Where declarations appear affects their scope and visibility
  - Rules are similar to those in Java
  - Declaration outside of any function are for global variables
    - e.g., just before the main routine

# Variable Declaration (2)

## Initialization

- ALWAYS initialize a variable before using it
  - Failure to do so in C is asking for trouble
  - The value of an uninitialized variables is undefined in the C standards
- Examples:

```
int count;          /* Set aside storage space for count */
count = 0;          /* Store 0 in count */
```

- This can be done at definition:

```
int count = 0;
double percent = 10.0,  rate = 0.56;
```

- Warning:  be careful about “out of range errors”

```
unsigned int value = -2500;
```

  - The C compiler does not detect this as an error
    - What do you suspect it does?

# Constants (1)

## Constants

- You can also declare variables as being constants
  - Use the const qualifier:

```
const double pi=3.1415926;
const int maxlength=2356;
const int val=(3*7+6)*5;
```

Note: simple computed values are allowed
• must be able to evaluate at *compile time*

  - Constants are useful for a number of reasons
    - Tells the reader of the code that a value does not change
      - Makes reading large pieces of code easier
    - Tells the compiler that a value does not change
      - The compiler can potentially compile faster code
- Use constants whenever appropriate
- NOTE: You will get errors with the cc compiler --- use the gcc compiler (newer)

# Constants (2)

## Preprocessor Constants

- These are an older form of constant which you still see
  - There is a potential for problems, so be careful using them!
- Generic Form:

  #define CONSTNAME literal
    - Generally make pre-processor constants all upper case (convention).
- Example:

  #define PI 3.14159
- What really happens
  - The C preprocessor runs before the compiler.
  - Every time it sees the token PI, it substitutes the value 3.14159.
  - The compiler is then run with this “pre-processed” C code.
- Why this is dangerous?
  - Hard to determine the value of a multiply-defined constant (which you are allowed to create)

# Constants (3)

- An example of constants in use:

```
#include <stdio.h>
#define GRAMS_PER_POUND 454
const int FARFARAWAY = 3000;
int main ()
{
   int weight_in_grams, weight_in_pounds;
   int year_of_birth, age_in_3000;
   printf ("Enter your weight in pounds: ");
   scanf  ("%d", &weight_in_pounds);
   printf ("Enter your year of birth: ");
   scanf  ("%d", &year_of_birth);
   weight_in_grams = weight_in_pounds * GRAMS_PER_POUND;
   age_in_3000 = FARFARAWAY - year_of_birth;
   printf ("Your weight in grams = %d\n", weight_in_grams);
   printf ("In 3000 you will be %d years old\n", age_in_3000);
   return 0;
}
```

# Literals in C

- Literals are representations of values of some types
  - C allows literal values for integer, character, and floating point types
- Integer literals
  - Just write the integer in base 10

    ```
    int y=-46;
    ```
  - We will discuss base 8 and base 16 literals later
- Character literals
  - Character literals are specified with a single character in single quotes

    ```
    char ch='a';
    ```
  - Special characters are specified with escape characters
    - Recall the discussion of ‘escaped’ characters with the shell
    - We will discuss these later
- Floating point literals
  - Just write the floating point number

    ```
    float PI=3.14159;
    ```
  - Can also use mantissa/exponent (scientific) notation

    ```
    double minusPItimes100 = -3.14159e2
    ```

# Flow Control

# Comments

- Comments: /* This is a comment */
  - Use them!
  - Comments should explain:
    - special cases
    - the use of functions (parameters, return values, purpose)
    - special tricks or things that are not obvious
  - explain *WHY* your code does things the what it does.

# More on Comments

- A bad comment:

```
...
i = i + 1;                    /* assign i+1 to the variable i */
...
```

- A better comment:

```
...
i = i + 1;                    /* increment the loop counter */
...
```

# C++ Comments

- A bad comment:

```
...
i = i + 1;                    // assign i+1 to the variable i
...
```

- A better comment:

```
...
i = i + 1;                    // increment the loop counter
...
```

# C Statements

- In the most general sense, a statement is a part of your program that can be executed.
- An expression is a statement.

```
a=a+1;
a--;
```

- A function call is also a statement.

```
printf("%d",a);
```

- Other statements ……
- C is a free form language, so you may type the statements in any style you feel comfortable:

```
a=
a+
1;a--;
```

line breaks can be anywhere

# Compound Statements

- Sequences of statements can be combined into one with {...}
- Much like Java:

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

- The C compiler treats the collection of these statements like they are a single statement.

# C Statements

## Some Suggestions

- DO: stay consistent with how you use whitespace
- DO: put block braces on their own line.
  - This makes the code easier to read.
- DO: line up block braces so that it is easy to find the beginning and end of a block.
- AVOID: spreading a single statement across multiple lines if there is no need.
  - Try to keep it on one line.

# The if Statement (1)

- Form 1:

```
if (expression)
        statement1;
next statement;
```

Execute statement1
if expression is non-zero
(i.e., it does not have to be exactly 1)

- Form 2:

```
if (expression)
        statement1;
else
        statement2;
next statement;
```

- Form 3:

```
if (expression)
        statement1;
else if (expression)
            statement2;
else
            statement3;
next statement;
```

# The if Statement (2)

- For Example:

```
#include <stdio.h>
int x,y;
int main ()
{
        printf ("\nInput an integer value for x: ");
        scanf ("%d", &x);
        printf ("\nInput an integer value for y: ");
        scanf ("%d",&y);
        if (x==y)
                printf ("x is equal to y\n");
        else if (x > y)
                printf ("x is greater than y\n");
        else
                printf ("x is smaller than y\n");
        return 0;
}
```

# The for Statement (1)

- The most important looping structure in C.
- Generic Form:

  *for (initial ; condition ; increment )*
  *statement*

- *initial*, *condition*, and *increment* are C expressions.
- For loops are executed as follows:

1. *initial* is evaluated. Usually an assignment statement.
2. *condition* is evaluated. Usually a relational expression.
3. If *condition* is false (i.e. 0), fall out of the loop (go to step 6.)
4. If *condition* is true (i.e. nonzero), execute *statement*
5. Execute *increment* and go back to step 2.
6. Next statement

# The for Statement (2)

## For statement examples

```
#include <stdio.h>
int main () {
   int count,x,y;
   int ctd;

   /* 1. simple counted for loop */
   for (count =1; count <=20; count++)
       printf ("%d\n", count);

   /* 2. for loop counting backwards */
   for (count = 100; count >0; count--) {
      x*=count;
      printf("count=%d x=%d\n", count,x);
   }

   /* 3. for loop counting by 5's */
   for (count=0; count<1000; count += 5)
         y=y+count;

   /* 4. initialization outside of loop */
   count = 1;
   for ( ; count < 1000; count++)
      printf("%d ", count);

   /* 5. very little need be in the for */
   count=1; ctd=1;
   for ( ; ctd; ) {
       printf("%d ", count);
       count++; ctd=count<1000;
   }

   /* 6. compound statements for
     initialization and increment */
   for (x=0, y=100; x<y; x++, y--) {
      printf("%d %d\n", x,y);
   }
   return 0;
}
```

# The for Statement (3)

- Nesting for Statements
  - for statements (and any other C statement) can go inside the loop of a for statement.
  - For example:

```
#include <stdio.h>
int main( ) {
   int rows=10, columns=20;
   int r, c;
   for ( r=rows ; r>0 ; r--)
   {
       for (c = columns; c>0; c--)
          printf ("X");
       printf ("\n");
    }
}
```

# The while Statement

- Generic Form

```
while (condition)
    statement
```

- Executes as expected:

1. condition is evaluated
2. If condition is false (i.e. 0), loop is exited (go to step 5)
3. If condition is true (i.e. nonzero), statement is executed
4. Go to step 1
5. Next statement

- Note:
  - for ( ; condition ; ) is equivalent to while (condition)
    stmt; stmt;
  - for (exp1; exp2; exp3) stmt;
    is equivalent to
    exp1;
    while(exp2) { stmt; exp3; }

# The do ... while Loop (1)

- Generic Form:

```
do
    statement
while (condition);
```

- Standard repeat until loop
  - Like a while loop, but with condition test at bottom.
  - Always executes at least once.
- The semantics of do...while:
  1. Execute statement
  2. Evaluate condition
  3. If condition is true go to step 1
  4. Next statement

# The do ... while Loop (2)

```
#include <stdio.h>
int get_menu_choice (void);
main()
{
  int choice;
  do
  {
    choice = get_menu_choice ();
    printf ("You chose %d\n",choice);
  } while(choice!=4);
  return 0;
}
```

```
/* simple function get_menu_choice */

int get_menu_choice (void)
{
  int selection = 0;
  do {
    printf ("\n");
    printf ("\n1 - Add a Record ");
    printf ("\n2 - Change a Record ");
    printf ("\n3 - Delete a Record ");
    printf ("\n4 - Quit ");
    printf ("\n\nEnter a selection: ");
    scanf  ("%d", &selection);
  } while ( selection<1 || selection>4);
  return selection;
}
```

# break and continue

- The flow of control in any loop can be changed through the use of the break and continue commands.
- The break command exits the loop immediately.
  - Useful for stopping on conditions not controlled in the loop condition.
  - For example:

```
for (x=0; x<10000; x++) {
   if ( x*x % 5==1) break;
   ... do some more work ...
}
```

  - Loop terminates if x*x % 5 == 1
- The continue command causes the next iteration of the loop to be started immediately.
  - For example:

```
for (x=0; x<10000; x++) {
   if (x*x % 5 == 1) continue;
   printf( "%d ", 1/ (x*x % 5 – 1) );
}
```

  - Don't execute loop when x*x % 5 == 1 (and avoid division by 0)

# Example: for and break Together

```
const int mycard=3;
int guess;
for(;;)
{
    printf("Guess my card:");
    scanf("%d",&guess);
    if(guess==mycard)
    {
        printf("Good guess!\n");
        break;
    }
    else
        printf("Try again.\n");
}
```

The notation for(;;) is used to create an infinite for loop. while(1) creates an infinite while loop instead.

To get out of an infinite loop like this one, we have to use the break statement.

# switch Statement

- Switch statement is used to do “multiple choices”.
- Generic form:

```
switch(expression)
{
    case constant_expr1 : statements
    case constant_expr2 : statements
    …
    case constant_exprk : statements
    default : statements
}
```

1. expression is evaluated.
2. The program jumps to the corresponding constant_expr.
3. All statements after the constant_expr are executed until a break (or goto, return) statement is encountered.

# Example: switch Statement

```
int a;
printf("1. Open file..\n");
printf("2. Save file.\n");
printf("3. Save as..\n");
printf("4. Quit.\n");
printf("Your choice:");
scanf("%d", &a);
if(a==1)
    open_file();
else if(a==2)
    save_file();
else if(a==3)
    save_as();
else if(a==4)   return 0;
else  return 1;
```

```
int a;
printf("1. Open file..\n");
printf("2. Save file.\n");
printf("3. Save as..\n");
printf("4. Quit.\n");
printf("Your choice:");
scanf("%d", &a);
switch(a)
{
    case 1: open_file();break;
    case 2: save_file();break;
    case 3: save_as();break;
    case 4: return 0;
    default: return 1;
}
```

# Jumping Out of Nested Loops -- goto

- The goto statement will jump to any point of your program.
- Use only if it is absolutely necessary (never in this course)

```
for(;;)
{
    ……
    while(…)
    {
        switch(…)
        {
            ……
            case … : goto finished;    /* finished is a label */
        }
    }
}
finished:    /* Jumped out from the nested loops */
```

**Expression and Operator**

Never jump into a loop!
Never jump backward!

# Functions

# Example

- int max(int a, int b);

```
int main(){
    int x;
    x = max(5,8);
    x = max(x,7);
}
```

```
int max(int a, int b){
        return a>b?a:b;
}
```

# What is a C Function?

- A function receives zero or more parameters, performs a specific task, and returns zero or one value.
- A function is invoked by its name and parameters.
  - No two functions have the same name in your C program.
  - No two functions have the same name AND parameter types in your C++ program.
  - The communication between the function and invoker is through the parameters and the return value.
- A function is independent:
  - It is “completely” self-contained.
  - It can be called at any places of your code and can be ported to another program.
- Functions make programs reusable and readable.

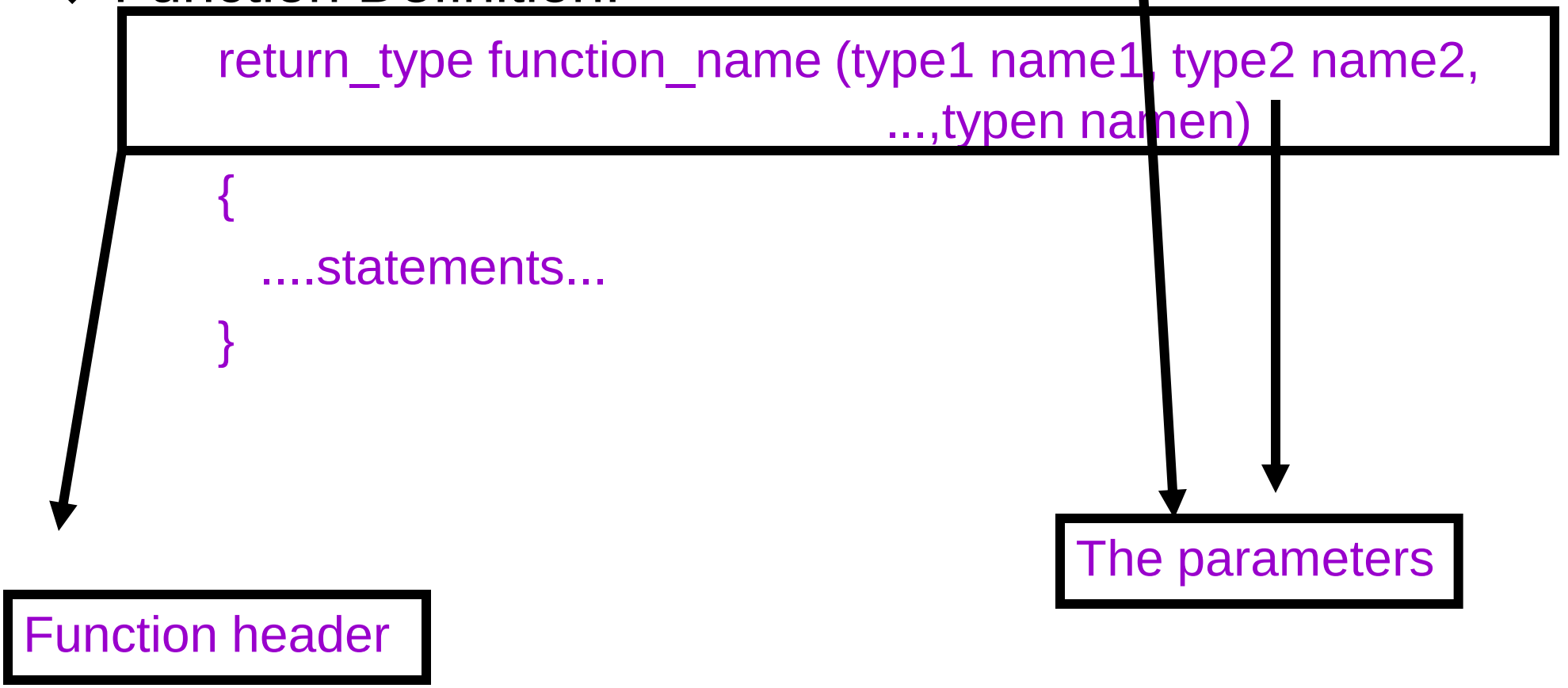
# Syntax

- Function Prototype:

```
return_type  function_name (type1  name1, type2  name2,
                            ..., typen  namen);
```

- Function Definition:

```
return_type function_name (type1 name1, type2 name2,
                           ...,typen namen)
{
   ....statements...
}
```



# Some Examples

- Function Prototype Examples

```
double squared (double number);
void print_report (int);
int get_menu_choice (void);
```

- Function Definition Examples

```
double squared (double number)
{
   return (number * number);
}

void print_report (int report_number)
{
   if (report_nmber == 1)
        printf("Printer Report 1");
     else
        printf("Not printing Report 1");
 }
```

void parameter list means it takes no parameters

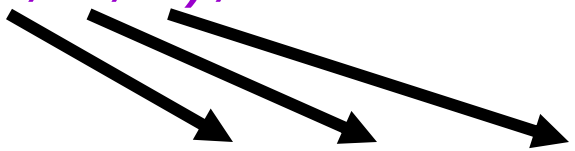
return type void means it returns nothing

# Passing Arguments

- Arguments are passed as in Java and Pascal
- Function call:

```
func1 (a, b, c);
```

- Function header



```
int func1 (int x, int y, int z)
```

  - Each argument can be any valid C expression that has a value:
  - For example:

```
x = func1(x+1,func1(2,3,4),5);
```

- Parameters x y z are initialized by the value of a b c
- Type conversions may occur if types do not match.

# Parameters are Passed by Value

- **All parameters are passed by value!!**
  - This means they are basically local variables initialized to the values that the function is called with.
  - They can be modified as you wish but these modifications will not be seen in the calling routine!

```
#include<stdio.h>
int twice(int x)
{
   x=x+x;
   return x;
}
int main()
{
   int x=10,y;
   y=twice(x);
   printf("%d,%d\n",x,y);
}
```

# Returning a Value

- To return a value from a C function you must explicitly return it with a return statement.
- Syntax:

```
return <expression>;
```

  - The expression can be any valid C expression that resolves to the type defined in the function header.
  - Type conversion may occur if type does not match.
  - Multiple return statements can be used within a single function (eg: inside an “if-then-else” statement…)

# Local Variables

- Local Variables

```
int func1 (int y)
{
    int a, b = 10;
    float rate;
    double cost = 12.55;
    .......
}
```

- Those variables declared “within” the function are considered “local variables”.
- They can only be used inside the function they were declared in, and not elsewhere.

# A Simple Example

```
#include <stdio.h>
int x=1;   /* global variable - bad! */
void demo(void);
int main() {
  int y=2;  /* local variable to main */
  printf ("\nBefore calling demo(), x = %d and y = %d.",x,y);
  demo();
  printf ("\nAfter calling demo(), x = %d and y = %d.\n",x,y);
  return 0;
}
void demo () {
  int x = 88, y =99;  /* local variables to demo */
  printf ("\nWithin demo(), x = %d and y = %d.",x,y);
}
```

# Placement of Functions

- For large programs
  - Manage related functions in a .c file
  - Write a .h file containing all the prototypes of the functions
  - #include the header file in the files that uses the functions.
- For small programs, use the following order in the only one file:
  - All prototypes
  - main() function
  - Other functions

- mymath.h

```
int min(int x,int y);
int max(int x,int y);
```

- mymath.c

```
int min(int x,int y)
{
    return x>y?y:x;
}
int max(int x,int y)
{
    return x>y?x:y;
}
```

# Recursion - An Example

```
unsigned int factorial(unsigned
    int a);


int main () {
  unsigned int f,x;
  printf("Enter value between 1
    & 8: ");
  scanf("%d", &x);
  if (x > 8 || x < 1)
    printf (”Illegal input!\n");
  else {
    f = factorial(x);
    printf ("%u factorial equals
    %u\n", x,f);
  }
}
```

```
unsigned int factorial (unsigned
    int a) {
  if (a==1)
    return 1;
  else {
    a *= factorial(a-1);
    return a;
  }
}
```

# Arrays

# Single-Dimensional Arrays

- Generic declaration:

  typename variablename[size]

  - typename is any type
  - variablename is any legal variable name
  - size is a number the compiler can figure out

- For example

  int a[10];

- Defines an array of ints with subscripts ranging from 0 to 9
- There are 10*sizeof(int) bytes of memory reserved for this array.


- You can use a[0]=10; x=a[2]; a[3]=a[2]; etc.
- You can use scanf("%d",&a[3]);

# Using Constants to Define Arrays

- It is useful to define arrays using constants:

```
#define  MONTHS 12
int  array [MONTHS];
```

- However, in ANSI C, you cannot

```
int n;
scanf("%d", &n);
int array[n];
```

- In GNU C, the variable length array is allowed.
- In ANSI C, the handling of variable length array is more complicated.

# Array-Bounds Checking

- C, unlike many languages, does NOT check array bounds subscripts during:
  - Compilation (some C compilers will check literals)
  - Runtime (bounds are never checked)
- If you access off the ends of any array, it will calculate the address it expects the data to be at, and then attempts to use it anyways
  - may get “something…”
  - may get a memory exception (segmentation fault, bus error, core dump error)
- It is the programmer’s responsibility to ensure that their programs are correctly written and debugged!
  - This does have some advantages but it does give you all the rope you need to hang yourself!

# Initializing Arrays

- Initialization of arrays can be done by a comma separated list following its definition.
- For example:

```
int array [4] = { 100, 200, 300, 400 };
```

  - This is equivalent to:

```
int array [4];
array[0] = 100;
array[1] = 200;
array[2] = 300;
array[3] = 400;
```

- You can also let the compiler figure out the array size for you:

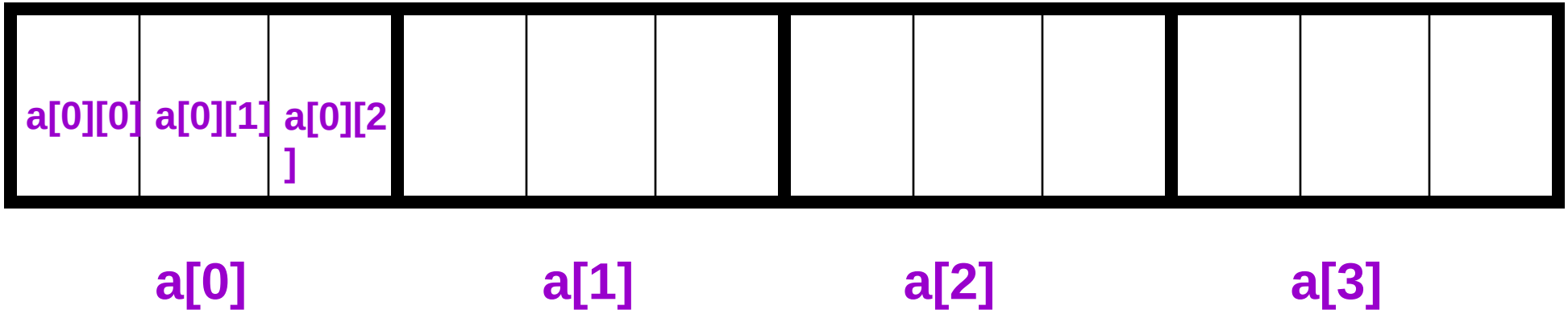
```
int array[] = { 100, 200, 300, 400};
```

# A Simple Example

```
#include <stdio.h>
int main() {
  float expenses[12]={10.3, 9, 7.5, 4.3, 10.5, 7.5, 7.5, 8, 9.9,
    10.2, 11.5, 7.8};
  int count,month;
  float total;
  for (month=0, total=0.0; month < 12; month++)
  {
       total+=expenses[month];
  }
  for (count=0; count < 12; count++)
      printf ("Month %d = %.2f K$\n", count+1, expenses[count]);
  printf("Total = %.2f K$,   Average = %.2f K$\n", total, total/12);
  return 0;
}
```

# Multidimensional Arrays

- Arrays in C can have virtually as many dimensions as you want.
- Definition is accomplished by adding additional subscripts when it is defined.
- For example:
  - int a [4] [3] ;
    - defines a two dimensional array
    - a is an array of int[3];
- In memory:



# Initializing Multidimensional Arrays

- The following initializes a[4][3]:

```
int a[4] [3] = { {1, 2, 3} , { 4, 5, 6} , {7, 8, 9} , {10, 11, 12} };
```

- Also can be done by:

```
int a[4] [3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
```

  - is equivalent to

```
a[0][0] = 1;
a[0][1] = 2;
a[0][2] = 3;
a[1][0] = 4;
...
a[3][2] = 12;
```

# An Example

```
#include <stdio.h>
#include <stdlib.h>
int main ()  {
    int random1[8][8];
    int a, b;
    for (a = 0; a < 8; a++)
        for (b = 0; b < 8; b++)
                    random1[a][b] = rand()%2;
    for (a = 0; a < 8; a++)
    {
        for (b = 0; b < 8; b++)
             printf ("%c " , random1[a][b] ? 'x' : 'o');
        printf("\n");
    }
   return 0;
}
```

# The value of the array name

```
#include <stdio.h>
int main(){
    int i;
    int a[3] = { 1, 2, 3 };
    printf( "a ? %d\n", a);
    printf( "a[0] ? %d\na[1] ? %d\na[2]
        ? %d\n", a[0], a[1], a[2]);
    printf( "&a[0] ? %d\n&a[1] ?
        %d\n&a[2] ? %d\n", &a[0],
        &a[1], &a[2]);

    printf( "\na[0] <- 4 \n");
    a[0] = 4;
    printf( "a ? %d\n", a);
    printf( "a[0] ? %d\na[1] ? %d\na[2]
        ? %d\n", a[0], a[1], a[2]);
    printf( "&a[0] ? %d\n&a[1] ?
        %d\n&a[2] ? %d\n\n", &a[0],
        &a[1], &a[2]);
    for (i=0; i<3; i++) {
        printf( "a[%d] <- ",i);
        scanf( "%d", &a[i]);
    }
    printf( "a ? %d\n", a);
    printf( "a[0] ? %d\na[1] ? %d\na[2]
        ? %d\n", a[0], a[1], a[2]);
    printf( "&a[0] ? %d\n&a[1] ?
        %d\n&a[2] ? %d\n", &a[0],
        &a[1], &a[2]);
}
```

- When the array name is used alone, its value is the address of the array (a pointer to its address).
- &a has no meaning if used in this program.

# Arrays as Function Parameters

- In this program, the array addresses (i.e., the values of the array names), are passed to the function inc_array().
- This does not conflict with the rule that “parameters are passed by values”.

```
void inc_array(int a[ ], int size)
{
    int i;
    for(i=0;i<size;i++)
    {
            a[i]++;
    }
}
```

```
void inc_array(int a[ ],int size);
main()
{
    int test[3]={1,2,3};
    int ary[4]={1,2,3,4};
    int i;
    inc_array(test,3);
    for(i=0;i<3;i++)
            printf("%d\n",test[i]);
    inc_array(ary,4);
    for(i=0;i<4;i++)
            printf("%d\n",ary[i]);
    return 0;
}
```

# An Example -- Sorting

```
void mysort(int a[ ],int size)
{
  int i,j,x;
  for(i=0; i<size; i++)
  {
    for(j=i; j>0; j--)
    {
      if(a[ j ] < a[ j-1])
      {  /* Change the order of a[ j ] and
      a[ j-1] */
          x=a[ j ];a[ j ]=a[ j-1]; a[j-1]=x;
      }
    }
  }
}
```

```
int main()
{
      int i;
      int tab[10] = {3,6,3,5,9,2,4,5,6,0};

      for(i=0;i<10;i++)
              printf("%d ",tab[i]);

      printf("\n");
      mysort(tab,10);

      for(i=0;i<10;i++)
              printf("%d ",tab[i]);

      printf("\n");
      return 0;
}
```

# Strings

# Strings are Character Arrays

- Strings in C are simply arrays of characters.
  - Example: `char s [10];`
- This is a ten (10) element array that can hold a character string consisting of ≤ 9 characters.
- This is because C does not know where the end of an array is at run time.
  - By convention, C uses a NULL character '\0' to terminate all strings in its library functions
- For example:

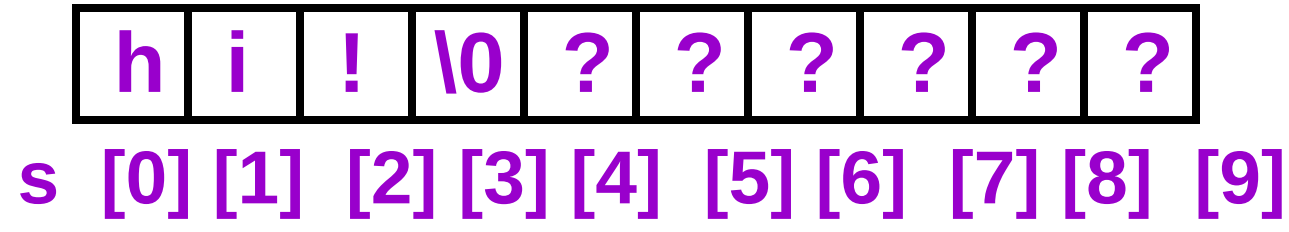
  ```
  char str [10] = {'u', 'n', 'l', 'x', '\0'};
  ```

- It's the string terminator (not the size of the array) that determines the length of the string.

# Accessing Individual Characters

- The first element of any array in C is at index 0. The second is at index 1, and so on ...

```
char s[10];
s[0] = 'h';
s[1] = 'i';
s[2] = '!';
s[3] = '\0';
```




- This notation can be used in all kinds of statements and expressions in C:
- For example:

```
c = s[1];
if (s[0] == '-') …
switch (s[1]) ...
```

# String Literals

- String literals are given as a string quoted by double quotes.
  - printf("Long long ago.");
- Initializing char array ...
  - char s[10]="unix";  /* s[4] is '\0';  */
  - char s[ ]="unix";  /* s has five elements */

# Printing with printf ( )

- Example:

```
char str[ ] = "A message to display";
printf ("%s\n", str);
```

- printf expects to receive a string as an additional parameter when it sees %s in the format string
  - Can be from a character array.
  - Can be another literal string.
  - Can be from a character pointer (more on this later).

- printf knows how much to print out because of the NULL character at the end of all strings.
  - When it finds a \0, it knows to stop.

# Example

```
char str[10]="unix and c";

 printf("%s", str);
 printf("\n");
 str[6]='\0';
 printf("%s", str);
 printf("\n");

 printf("\n");
 printf(str);
 printf("\n");
 str[2]='%';
 printf(str);
 printf("\n");
```

# Printing with puts( )

- The puts function is a much simpler output function than printf for string printing.
- Prototype of puts is defined in stdio.h

  int puts(const char * str)

  – This is more efficient than printf

    - Because your program doesn't need to analyze the format string at run-time.

- For example:

```
char sentence[] = "The quick brown fox\n";
puts(sentence);
```

- Prints out:

```
The quick brown fox
```

# Inputting Strings with gets( )

- gets( ) gets a line from the standard input.
- The prototype is defined in stdio.h
  char *gets(char *str)
  - str is a pointer to the space where gets will store the line to, or a character array.
  - Returns NULL upon failure. Otherwise, it returns str.

```
char your_line[100];
printf("Enter a line:\n");
gets(your_line);
puts("Your input follows:\n");
puts(your_line);
```

  - You can overflow your string buffer, so be careful!

# Inputting Strings with scanf ( )

- To read a string include:
  - %s  scans up to but not including the “next” white space character
  - %***n***s scans the next ***n*** characters or up to the next white space character, whichever comes first
- Example:

```
scanf ("%s%s%s", s1, s2, s3);
scanf ("%2s%2s%2s", s1, s2, s3);
```

  - Note:  No ampersand(&) when inputting strings into character arrays!  (We’ll explain why later …)
- Difference between gets
  - gets( )  read a line
  - scanf("%s",…) read up to the next space

# An Example

```
#include <stdio.h>
int main () {
    char lname[81], fname[81];
    int  count, id_num;
    puts ("Enter the last name, firstname, ID number
    separated");
    puts ("by spaces, then press Enter \n");
    count = scanf ("%s%s%d", lname, fname,&id_num);
    printf ("%d items entered: %s %s %d\n",
                    count,fname,lname,id_num);
    return 0;
}
```

# The C String Library

- String functions are provided in an ANSI standard string library.
  - Access this through the include file:

    #include <string.h>
  - Includes functions such as:
    - Computing length of string
    - Copying strings
    - Concatenating strings
  - This library is guaranteed to be there in any ANSI standard implementation of C.

# strlen

- strlen returns the length of a NULL terminated character string:

```
size_t  strlen (char * str) ;
```

- Defined in string.h
- size_t
  - A type defined in string.h that is equivalent to an unsigned int
- char  *str
  - Points to a series of characters or is a character array ending with '\0'
  - The following code has a problem!

```
char a[5]={‘a’, ’b’, ’c’, ’d’, ’e’};
strlen(a);
```

# strcpy

- Copying a string comes in the form:

  char *strcpy (char * destination, char * source);

- A copy of source is made at destination
  - source should be NULL terminated
  - destination should have enough room (its length should be at least the size of source)
- The return value also points at the destination.

# strcat

- Included in string.h and comes in the form:

  char * strcat (char * str1, char * str2);

  - Appends a copy of str2 to the end of str1
  - A pointer equal to str1 is returned

- Ensure that str1 has sufficient space for the concatenated string!
  - Array index out of range will be the most popular bug in your C programming career.

# Example

```
#include <string.h>
#include <stdio.h>
int main() {
    char str1[27] = "abc";
    char str2[100];
    printf("%d\n",strlen(str1));
    strcpy(str2,str1);
    puts(str2);
    puts("\n");
    strcat(str2,str1);
    puts(str2);
}
```

# Comparing Strings

- C strings can be compared for equality or inequality
- If they are equal - they are ASCII identical
- If they are unequal the comparison function will return an int that is interpreted as:

  < 0 : str1 is less than str2

  0 : str1 is equal to str2

  > 0 : str1 is greater than str2

# strcmp

- Four basic comparison functions:

  int strcmp (char *str1, char *str2) ;

  - Does an ASCII comparison one char at a time until a difference is found between two chars
    - Return value is as stated before
  - If both strings reach a '\0' at the same time, they are considered equal.

  int strncmp (char *str1, char * str2, size_t n);

  - Compares n chars of str1 and str2
    - Continues until n chars are compared or
    - The end of str1or str2 is encountered

  – Also have strcasecmp() and strncasecmp() which do the same as above, but ignore case in letters.

# Example

- An Example - strncmp

```
int main() {
   char str1[] = "The first string.";
   char str2[] = "The second string.";
   size_t  n, x;
   printf("%d\n", strncmp(str1,str2,4) );
   printf("%d\n", strncmp(str1,str2,5) );
}
```

# Searching Strings (1)

- There are a number of searching functions:
  - char * strchr (char * str, int ch) ;
    - strchr search str until ch is found or NULL character is found instead.
    - If found, a (non-NULL) pointer to ch is returned.
      - Otherwise, NULL is returned instead.
  - You can determine its location (index) in the string by:
    - Subtracting the value returned from the address of the start of the string
      - More pointer arithmetic … more on this later!

# Example

Example use of strchr:

```
#include<stdio.h>
#include<string.h>
int main() {
   char ch='b', buf[80];
   strcpy(buf, "The quick brown fox");
   if (strchr(buf,ch) == NULL)
       printf ("The character %c was not found.\n",ch);
   else
       printf ("The character %c was found at position
                %d\n", ch, strchr(buf,ch)-buf+1);
}
```

# Searching Strings (2)

- Another string searching function:

  char * strstr (char * str, char * query) ;

  - strstr searches str until query is found or a NULL character is found instead.
  - If found, a (non-NULL) pointer to str is returned.
    - Otherwise, NULL is returned instead.

# sprintf

```
#include <stdio.h>

int sprintf( char *s, const char *format, …);
```

- Instead of printing to the stdin with printf(…), sprintf prints to a string.
- Very useful for formatting a string, or when one needs to convert integers or floating point numbers to strings.
- There is also a sscanf for formatted input from a string in the same way scanf works.

# Example:

```
#include <stdio.h>
#include <string.h>
int main()
{
      char result[100];
      sprintf(result, "%f", (float)17/37 );
      if (strstr(result, "45") != NULL)
        printf("The digit sequence 45 is in 17
                    divided by 37. \n");
      return 0;
}
```

# Converting Strings to Numbers (1)

- Contained in <stdlib.h> and are often used

  int atoi (char *ptr);

  - Takes a character string and converts it to an integer.
  - White space and + or - are OK.
  - Starts at beginning and continues until something non-convertible is encountered.

- Some examples:

| String | Value returned |
|---|---|
| "157" | 157 |
| "-1.6" | -1 |
| "+50x" | 50 |
| "twelve" | 0 |
| "x506" | 0 |

# Converting Strings to Numbers (2)

long atol (char *ptr) ;

- Same as atoi except it returns a long.

double atof (char * str);

- Handles digits 0-9.
- A decimal point.
- An exponent indicator (e or E).
- If no characters are convertible a 0 is returned.

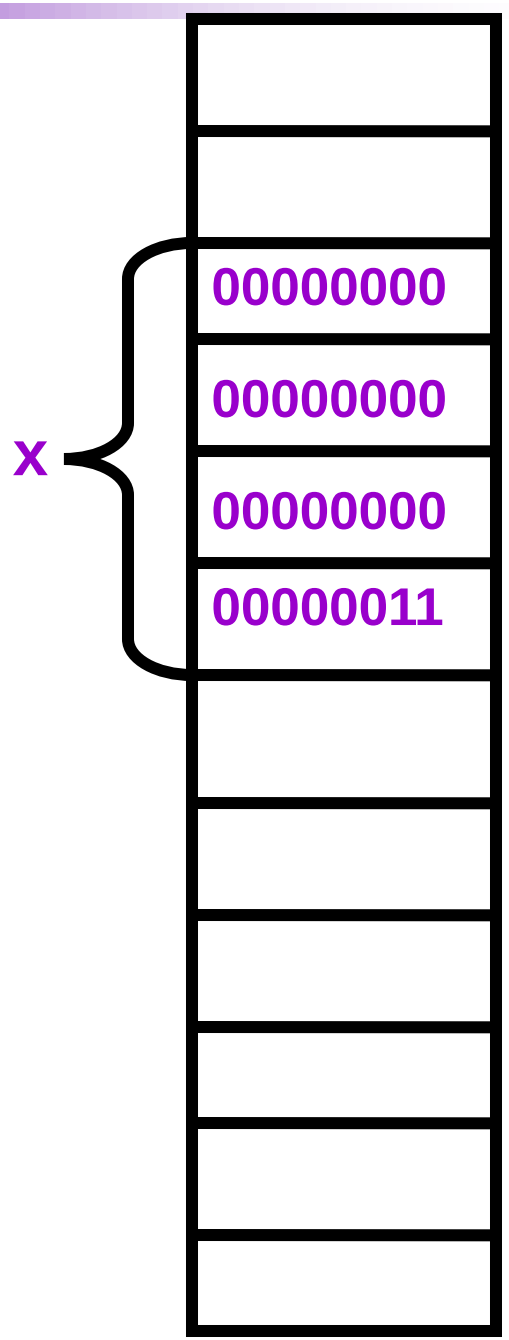
◆ Examples:

| String | Value returned |
|---|---|
| "12" | 12.000000 |
| "-0.123" | -0.123000 |
| "123E+3" | 123000.000000 |
| "123.1e-5" | 0.001231 |

# Pointers

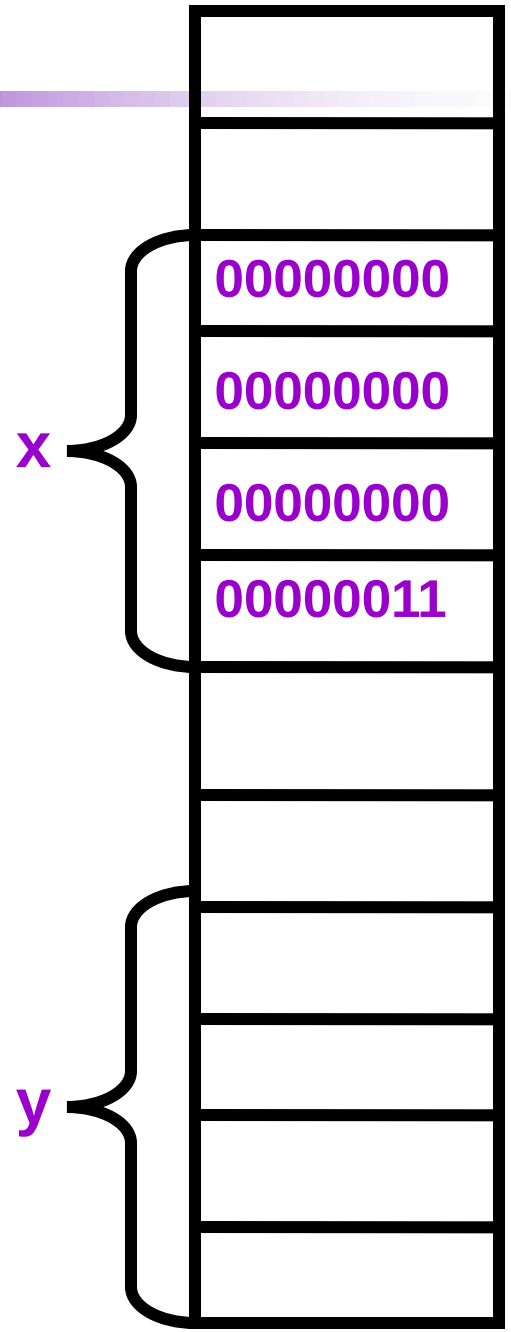
# Pointer Fundamentals

- When a variable is defined the compiler (linker/loader actually) allocates a real memory address for the variable.
  - int x; will allocate 4 bytes in the main memory, which will be used to store an integer value.
- When a value is assigned to a variable, the value is actually placed to the memory that was allocated.
  - x=3; will store integer 3 in the 4 bytes of memory.



# Pointers

- When the value of a variable is used, the contents in the memory are used.
  - y=x;  will read the contents in the 4 bytes of memory, and then assign it to variable y.
- &x can get the address of x. (referencing operator &)
- The address can be passed to a function:
  - scanf("%d", &x);
- The address can also be stored in a variable ……



# Pointers

- To declare a pointer variable

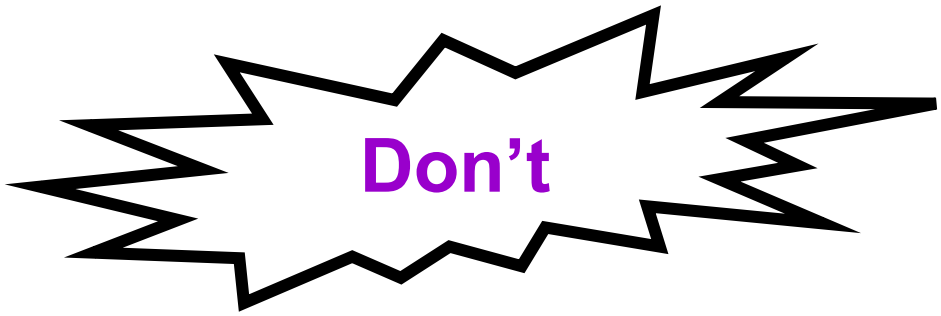
  type * pointername;

- For example:
  - int * p1;  p1 is a variable that tends to point to an integer, (or p1 is a int pointer)
  - char *p2;
  - unsigned int * p3;
- p1 = &x;  /* Store the address in p1 */
- scanf("%d", p1);  /* i.e. scanf("%d",&x); */
- p2 = &x;  /* Will get warning message */

# Initializing Pointers

- Like other variables, always initialize pointers before using them!!!
- For example:

```
int main(){
   int x;
   int *p;
   scanf("%d",p);   /*                                  */
   p = &x;
   scanf("%d",p);  /* Correct */
}
```



# Using Pointers

- You can use pointers to access the values of other variables, *i.e.* the contents of the memory for other variables.
- To do this, use the * operator (dereferencing operator).
  - Depending on different context, * has different meanings.
- For example:

```
int n, m=3, *p;
p=&m;
n=*p;
printf("%d\n", n);
printf("%d\n",*p);
```

# An Example

```
int m=3, n=100, *p;
p=&m;
printf("m is %d\n",*p);
m++;
printf("now m is %d\n",*p);
p=&n;
printf("n is %d\n",*p);
*p=500;     /* *p is at the left of "="  */
printf("now n is %d\n", n);
```

# Pointers as Function Parameters

- Sometimes, you want a function to assign a value to a variable.
  - e.g. scanf()
- E.g. you want a function that computes the minimum AND maximum numbers in 2 integers.
- Method 1, use two global variables.
  - In the function, assign the minimum and maximum numbers to the two global variables.
  - When the function returns, the calling function can read the minimum and maximum numbers from the two global variables.
- This is bad because the function is not reusable.

# Pointers as Function Parameters

- Instead, we use the following function

```
void min_max(int a, int b,
        int *min, int *max){
    if(a>b){
        *max=a;
        *min=b;
    }
    else{
        *max=b;
        *min=a;
    }
}
```

```
int main()
{
    int x,y;
    int small,big;
    printf("Two integers: ");
    scanf("%d %d", &x, &y);
    min_max(x,y,&small,&big);
    printf("%d <= %d", small,
    big);
    return 0;
}
```

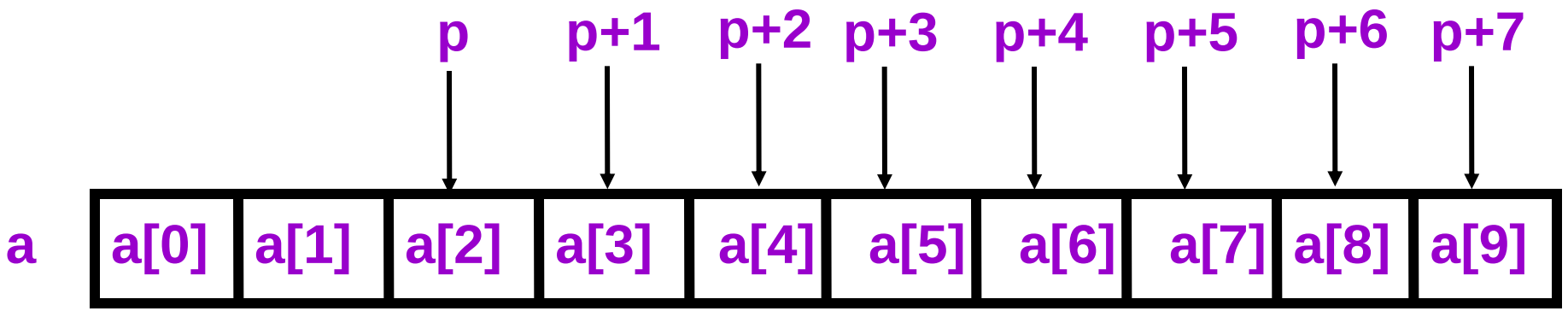
# Pointer Arithmetic (1)

**When a pointer variable points to an array element, there is a notion of adding or subtracting an integer to/from the pointer.**

```
int a[ 10 ], *p;
p = &a[2];
*p = 10;
*(p+1) = 10;
printf("%d", *(p+3));
```

```
int a[ 10 ], *p;


a[2] = 10;
a[3] = 10;
printf("%d", a[5]);
```



# Pointer Arithmetic (2)

- More examples:

```
int a[10], *p, *q;
p = &a[2];
q = p + 3;          /* q points to a[5] now */
p = q – 1;          /*  p points to a[4] now */
p++;                /*  p points to a[5] now */
p--;                /* p points to a[4] now */
*p = 123;           /*  a[4] = 123 */
*q = *p;            /*   a[5] = a[4]   */
q = p;              /*   q points to a[4] now */
scanf("%d", q)     /* scanf("%d", &a[4])  */
```

# Pointer Arithmetic (3)

- If two pointers point to elements of a same array, then there are notions of subtraction and comparisons between the two pointers.

```
int a[10], *p, *q , i;
p = &a[2];
q = &a[5];
i = q - p;        /* i is 3*/
i = p - q;        /* i is -3 */
a[2] = a[5] = 0;
i = *p - *q;      /* i = a[2] – a[5]  */
p < q;            /* true */
p == q;           /* false */
p != q;           /* true */
```

# Pointers and Arrays

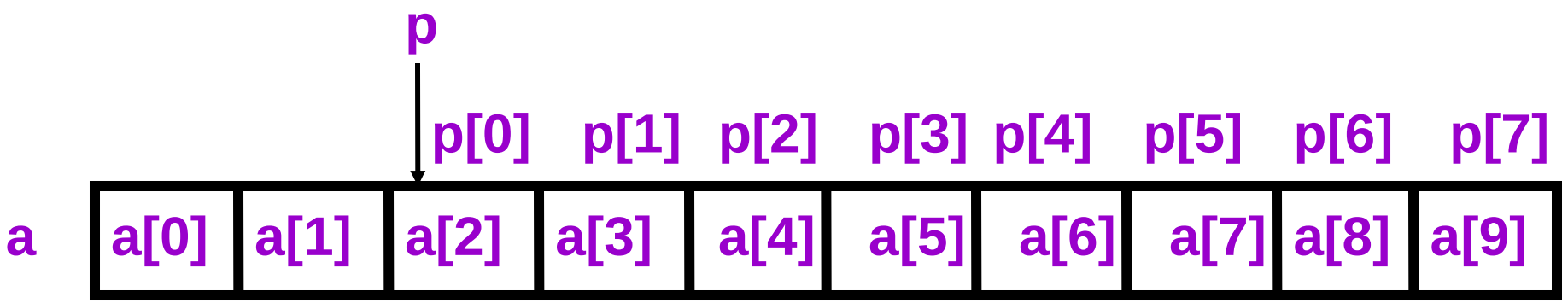
- Recall that the value of an array name is also an address.
- In fact, pointers and array names can be used interchangeably in many (but not all) cases.
  - E.g. int n, *p; p=&n;
  - n=1; *p = 1; p[0] = 1;
- The major differences are:
  - Array names come with valid spaces where they “point” to. And you cannot “point” the names to other places.
  - Pointers do not point to valid space when they are created. You have to point them to some valid space (initialization).

# Using Pointers to Access Array Elements

```
int a[ 10 ], *p;
p = &a[2];
p[0] = 10;
p[1] = 10;
printf("%d", p[3]);
```

```
int a[ 10 ], *p;


a[2] = 10;
a[3] = 10;
printf("%d", a[5]);
```



# An Array Name is Like a Constant Pointer

- Array name is like a constant pointer which points to the first element of the array.

```
int a[10], *p, *q;
p = a;              /* p = &a[0] */
q = a + 3;          /* q = &a[0] + 3 */
a ++;               /*  illegal !!! */
```

- Therefore, you can “pass an array” to a function. Actually, the address of the first element is passed.

```
int a[ ] = { 5, 7, 8 , 2, 3 };
sum( a, 5 );  /* Equal to sum(&a[0],5)  */
…….
```

# An Example

```
/* Sum – sum up the ints in
   the given array */
int sum(int *ary, int size)
{
   int i, s;
   for(i = 0, s=0; i<size;i++){
         s+=ary[i];
   }
   return s;
}
```

```
/* In another function
   */
   int a[1000],x;
   ……
   x=
   sum(&a[100],50);
   /* This sums up
   a[100], a[101], …,
   a[149] */
```

# Allocating Memory for a Pointer (1)

- The following program is wrong!

```
#include <stdio.h>
int main()
{
    int *p;
    scanf("%d",p);
    return 0;
}
```

- This one is correct:

```
#include <stdio.h>
int main()
{
    int *p;
    int a;
    p = &a;
    scanf("%d",p);
    return 0;
}
```

# Allocating Memory for a Pointer (2)

- There is another way to allocate memory so the pointer can point to something:

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int *p;
    p = (int *) malloc( sizeof(int) );   /* Allocate 4 bytes */
    scanf("%d", p);
    printf("%d", *p);
    free(p);        /* This returns the memory to the system.*/
                    /* Important !!! */
}
```

# Allocating Memory for a Pointer (3)

- Prototypes of malloc() and free() are defined in stdlib.h

```
void * malloc(size_t number_of_bytes);
void free(void * p);
```

- You can use malloc and free to dynamically allocate and release the memory;

```
int *p;
p = (int *) malloc(1000 * sizeof(int) );
for(i=0; i<1000; i++)
    p[i] = i;
p[1000]=3;   /* Wrong! */
free(p);
p[0]=5;   /* Wrong! */
```

# An Example – Finding Prime Numbers

```
#include <stdio.h>
#include <stdlib.h>
/* Print out all prime
numbers which are less
than m */
void print_prime( int m )
{
  int i,j;
  int * ary = (int *) malloc( m * sizeof(int));
  if (ary==NULL) exit -1;
  for(i=0;i<m;i++)
    ary[i]=1;
  /* Assume all integers between 0 and m-1 are
prime */
  ary[0]=ary[1]=0;
  /* Note that in fact 0 and 1 are not prime */
  for(i=3;i<m;i++){
    for( j=2; j<i; j++)
      if(ary[ i ] && i%j==0){
ary[i]=0;
break;
      }
  }
  for(i=0;i<m;i++)
    if(ary[i]) printf("%d ", i);
  free( ary );
  printf("\n");
}

int main() {
  int m;
  printf("m = ");
  scanf("%d", &m);
  printf("\n");
  print_prime(m);
  return 0;
}
```