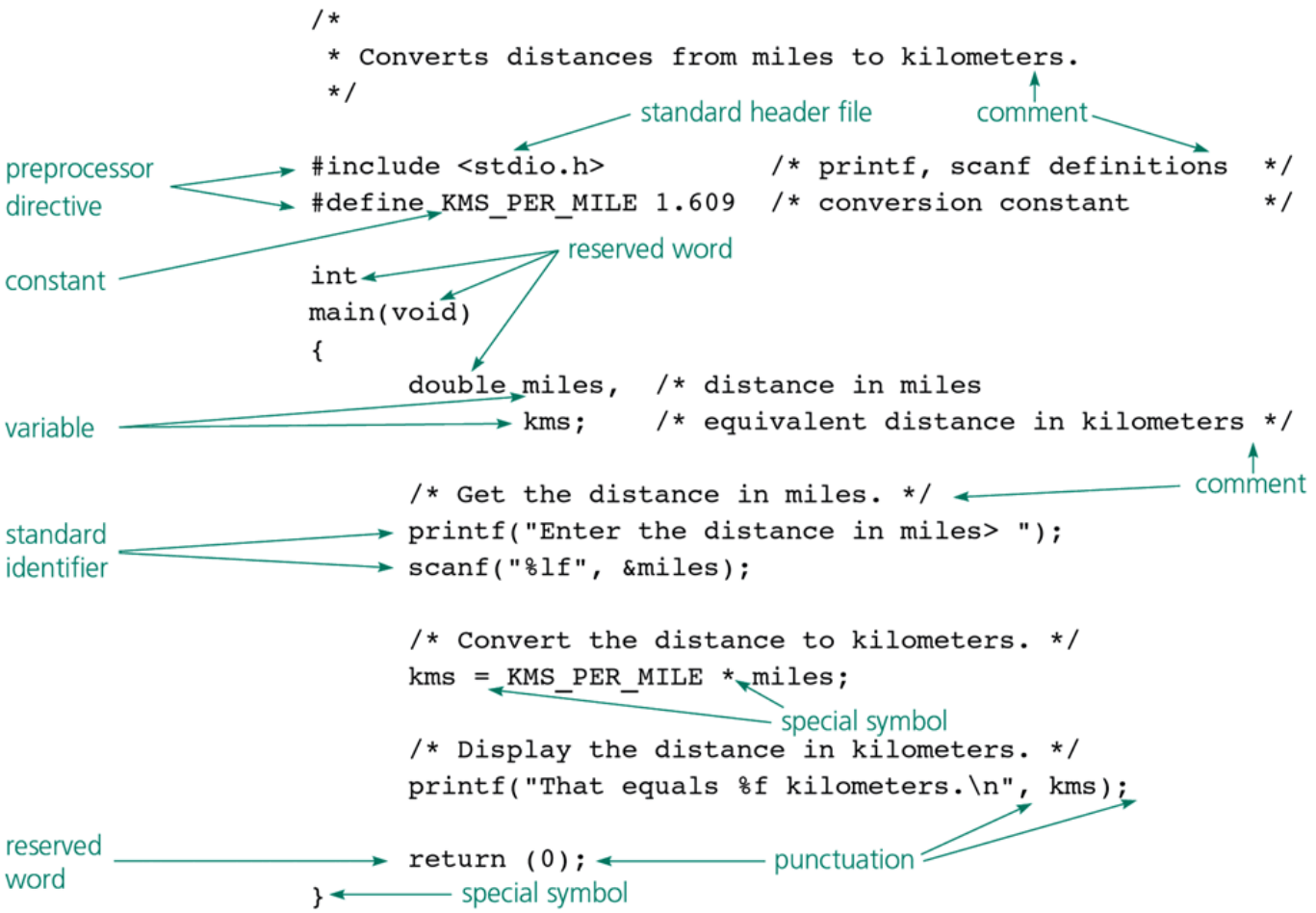
C Program Development and Debugging under Unix

C – Basic Elements



C - Basic Types

Type (32 bit)	Smallest Value	Largest Value
short int	-32,768(-2^{15})	32,767(2^{15}-1)
unsigned short int	0	65,535(2^{16}-1)
int	-2,147,483,648(-2^{31})	2,147,483,648(2^{31}-1)
unsigned int	0	4,294,967,295
long int	-2,147,483,648(-2^{31})	2,147,483,648(2^{31}-1)
unsigned long int	0	4,294,967,295

C - Floating Types

float	single-precision floating-point
double	double-precision floating-point
long double	extended-precision floating-point

Type	Smallest Positive Value	Largest Value	Precision
float	$1.17*10^{-38}$	$3.40*10^{38}$	6 digits
double	$2.22*10^{-308}$	$1.79*10^{308}$	15 digits

```
double x;
scanf("%lf", &x);
printf("%lf", x);
```

```
long double x;
scanf("%Lf", &x);
printf("%Lf", x);
```

C - Character Types

```
char ch;
int i;

ch = ‘a’

i = ‘a’;                /* i is now 97 */

ch = 65;                /* ch is now ‘A’ */
ch = ch + 1;            /* ch is now ‘B’ */
ch++;                   /* ch is now ‘C’ */

if(‘a’ <= ch && ch <= ‘z’)

for(ch = ‘A’; ch <= ‘Z’; ch++)
```

C - Char Type

‘a‘, ‘\t’, ‘\n’, ‘\0’, etc. are character constants

strings: character arrays

- (see <string.h> for string functions)
- "I am a string"
- always null (‘\0’) terminated.
- 'x' is different from "x"

Reading Input from Console

```
//reading input from console
#include <stdio.h>
int main()
{
   int num1;
   int num2;
   printf( "Please enter two numbers: " );
   scanf( "%d %d", &num1,&num2 );
   printf( "You entered %d %d", num1, num2 );
   return 0;
}
```

C - Control Flow

- blocks: { ... }
- if (**expr**) **stmt**;
- if (**expr**) **stmt1** else **stmt2**;
- switch (**expr**) {case ... default }
- while (**expr**) stmt;
- for (**expr1**;**expr2**;**expr3**) **stmt**;
- do **stmt** while **expr**;
- break; continue (only for loops);
- goto label;

C - Loops(for)

```
#include <stdio.h>
int main()
{
   int x;
   /* The loop goes while x < 10, and x increases by one every loop*/
   for ( x = 0; x < 10; x++ )
   {
      /* Keep in mind that the loop condition checks
         the conditional statement before it loops again.
         consequently, when x equals 10 the loop breaks.
         x is updated before the condition is checked. */
         printf( "%d\n", x );
   }
   return 0;
}
```

C - Loops(while)

```
#include <stdio.h>

int main()
{
  int x = 0;  /* Don't forget to declare variables */
  while ( x < 10 )
  {   /* While x is less than 10 */
      printf( "%d\n", x );
      x++;    /* Update x so the condition can be met eventually */
  }
  return 0;
}
```

C - Loops(do while)

```
#include <stdio.h>
int main()
{
  int x;
  x = 0;
  do
  {
    /* "Hello, world!" is printed at least one time
      even though the condition is false*/
     printf( "%d\n", x );
    x++;
  } while ( x != 10 );
  return 0;
}
```

C - function

```
#include <stdio.h>

//function declaration
int mult ( int x, int y );
int main()
{
  int x;
  int y;
  printf( "Please input two numbers to be multiplied: " );
  scanf( "%d", &x );
  scanf( "%d", &y );
  printf( "The product of your two numbers is %d\n", mult( x, y ) );
  return 0;
}

//define the function body
//return value: int
//utility: return the multiplication of two integer values
//parameters: take two int parameters
int mult (int x, int y)
{
  return x * y;
}
```

```
#include <stdio.h>
//function declaration, need to define the function body in other places
void playgame();
void loadgame();
void playmultiplayer();
int main()
{
   int input;
   printf( "1. Play game\n" );
   printf( "2. Load game\n" );
   printf( "3. Play multiplayer\n" );
   printf( "4. Exit\n" );
   printf( "Selection: " );
   scanf( "%d", &input );
   switch ( input ) {
      case 1:          /* Note the colon, not a semicolon */
         playgame();
         break;        //don't forget the break in each case
      case 2:
         loadgame();
         break;
      case 3:
         playmultiplayer();
         break;
      case 4:
         printf( "Thanks for playing!\n" );
         break;
      default:
         printf( "Bad input, quitting!\n" );
         break;
   }
   return 0;
}
```

C - struct

```
#include <stdio.h>
//group things together
struct database {
  int id_number;
  int age;
  float salary;
};

int main()
{
  struct database employee;
  employee.age = 22;
  employee.id_number = 1;
  employee.salary = 12000.21;
}
```

C - Type Definitions

```
typedef int BOOL
BOOL flag;   /*  same as int flag; */

typedef struct {int age; char *name} person;
person people;
```

Definition – *Array*

- A collection of objects of the *same type* stored contiguously in memory under one name
  - May be type of any kind of variable
  - May even be collection of arrays!
- For ease of access to any member of array
- For passing to functions as a group

Examples

- **`int A[10]`**
  - An array of ten integers
  - **`A[0]`**, **`A[1]`**, …, **`A[9]`**
- **`double B[20]`**
  - An array of twenty long floating point numbers
  - **`B[0]`**, **`B[1]`**, …, **`B[19]`**
- Arrays of **`structs`**, **`unions`**, **`pointers`**, etc., are also allowed
- Array indexes *always* start at zero in *C*

Examples (continued)

- **`int C[]`**
  - An array of an unknown number of integers (allowable in a parameter of a function)
  - **`C[0]`**, **`C[1]`**, …, **`C[`***max-1*`]`**
- **`int D[10][20]`**
  - An array of ten rows, each of which is an array of twenty integers
  - **`D[0][0]`**, **`D[0][1]`**, …, **`D[1][0]`**, **`D[1][1]`**, …, **`D[9][19]`**
  - Not used so often as arrays of pointers

Array Element

- May be used wherever a variable of the same type may be used
  - In an expression (including arguments)
  - On left side of assignment

- Examples:–

```
A[3] = x + y;
x = y - A[3];
z = sin(A[i]) + cos(B[j]);
```

Array Elements (continued)

- Generic form:–
  - *ArrayName*[*integer-expression*]
  - *ArrayName*[*integer-expression*] [*integer-expression*]
  - Same type as the underlying type of the array
- Definition:– *Array Index* – the expression between the square brackets

Array Elements (continued)

- Array elements are commonly used in loops
- E.g.,

```
for(i=0; i < max; i++)
  A[i] = i*i;

sum = 0; for(j=0; j < max; j++)
  sum += B[j];
```

Caution! Caution! Caution!

- It is the programmer's responsibility to avoid indexing off the end of an array
  - *Likely* to corrupt data
  - May cause a *segmentation fault*
  - Could expose system to a *security hole!*
- *C* does NOT check *array bounds*
  - I.e., whether index points to an element within the array
  - Might be high (beyond the end) or negative (before the array starts)

Single-module Programs

- Let's examine a C program that performs a simple task: reversing a string.
- We will learn how to write, compile, link, and execute a program that solves the problem using a single source file.
  - It's better to split a large program up into several independent modules. (will be discussed later)
- A source code listing of the first version of the reverse program is next presented.

```
1  /*  reverse.c  */
2  #include <stdio.h>
3  #include <string.h>
4  /*  Function prototype  */
5  void reverse (char before[], char after[]);
6
7  /*********************************************/
8  int main()
9  {
10   char str[100];   /* buffer to hold reversed string */
11   reverse("cat",str);  /* reverse the string "cat"  */
12   printf ("reverse(\"cat\") = %s\n", str);
13   reverse("noon",str);
14   printf ("reverse(\"noon\") = %s\n", str);
15 }
16
17 /*********************************************/
18 void reverse (char before[], char after[])
19 {
20   int i,j,len;
21
22   len = strlen(before);
23   i=0;
24   for (j=len-1; j>=0; j--)
25   {
26     after[i] = before[j];
27     i++;
28   }
29   after[len] = '\0';
30 }
```

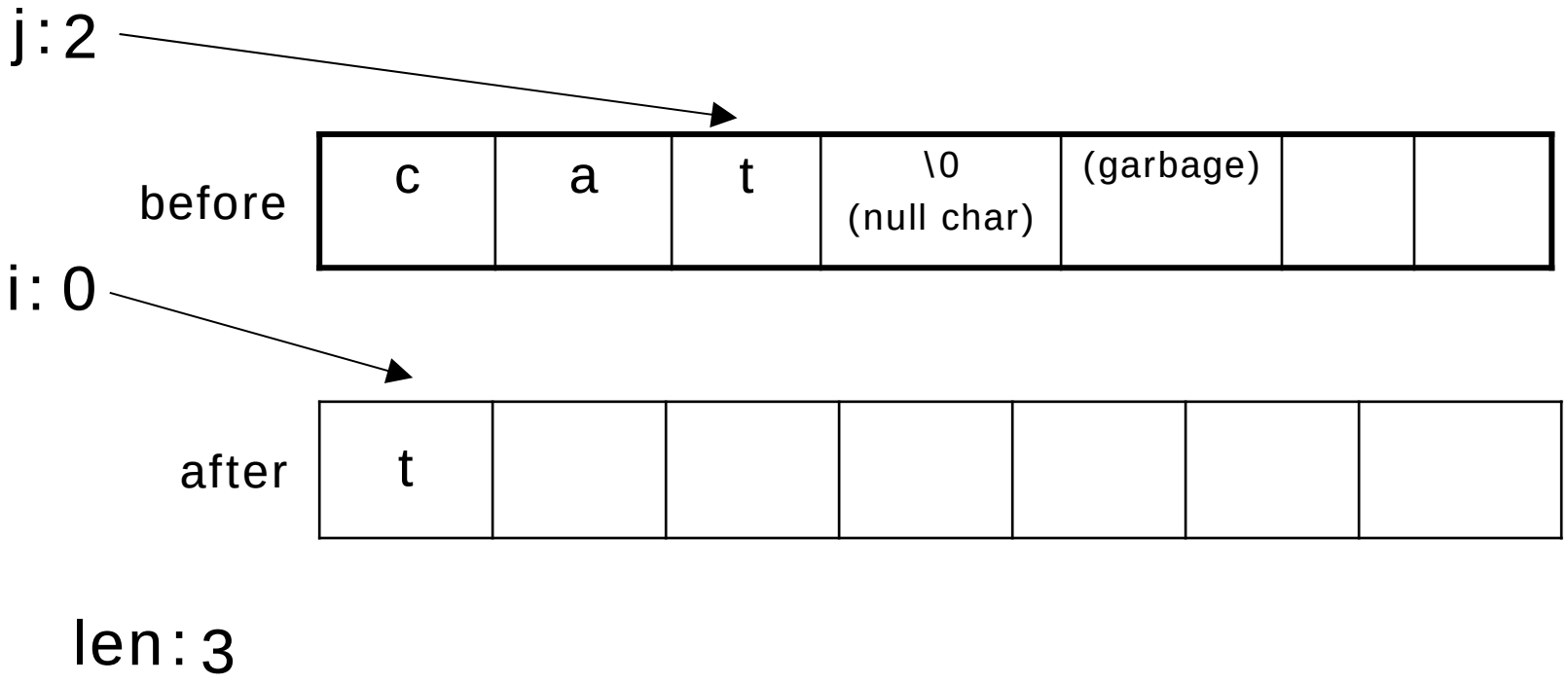
C String - Revision

- In C, a string is represented as an array of characters
- For example:

  “noon” is represented as:

0	1	2	3	4	5	
n	o	o	n	\0 (null char)	X (garbage)	

“Reverse” Function Logic



Syntax and Semantics

- The *syntax rules* of a language define how we can put together symbols, reserved words, and identifiers to make a valid program
- The *semantics* of a program statement define what that statement means (its purpose or role in a program)
- A program that is syntactically correct is not necessarily logically (semantically) correct
- A program will always do what we tell it to do, not only what we *intend* to tell it to do

Program Errors

- A program can have three types of errors
- The compiler will find syntax errors and other basic problems (*compile-time errors*)
  - If compile-time errors exist, an executable version of the program is not created
- A problem can occur during program execution, such as trying to divide by zero, which causes a program to terminate abnormally (*run-time errors*)
- A program may run, but produce incorrect results, perhaps using an incorrect formula (*logical errors*)

Program Errors

- In Unix, many runtime errors and logical errors only provide the following error messages:
  - Segmentation fault
  - Bus error
- The above messages do not provide good hints for the causes of the error
  - In other words, it is almost not useful to rely on this message for debugging
- Therefore, debugging techniques or tools are needed
  - Inserting “printing” statements to display the content of variables
  - Use a debugger utility

Debugging by Inserting “printing” Statements

- A good way for debugging is to insert “printing” statements in the program.
- The purpose is to print the content of some important variables to see if the execution follows our design.
- Suppose we inserted the following line after the statement i++ in the reverse function:

  printf(“i=%d j=%d\n”,i,j)

- The execution would look like:

```
sepc92:> ./reverse                    ... run the executable “reverse”
i = 1   j = 2
i = 2   j = 1
i = 3   j = 0
reverse (“cat”) = tac
i = 1   j = 3
     :
```

Debugging by Inserting "printing" Statements (cont)

- When the output text to the screen is very long, we can *re-direct* the output text to a file instead of the screen so that we can examine the content in a more convenient way.
- To re-direct in Unix the output from the screen to a text file, we can use the character > meaning that the text output is re-directed from the system output (screen) to a specified text file.
- In the following example, it is re-directed to a text file called debug.txt

```
sepc92:> ./reverse > debug.txt        … run the executable "reverse"
sepc92:> more debug.txt
i = 1   j = 2
i = 2   j = 1
i = 3   j = 0
reverse ("cat") = tac
i = 1   j = 3
     :
```

The Unix Debugger : gdb

- **Preparing a Program for Debugging**

  To debug a program, it must have been compiled using the -g option to gcc, which places debugging information into the object module.

  sepc92:> gcc –g –o reverse reverse.c

- The UNIX debugger, **gdb,** allows you to debug a program symbolically. It includes the following facilities:
  - single stepping
  - breakpoints
  - editing from within the debugger
  - accessing and modifying variables
  - searching for functions
  - tracing

The Unix Debugger : gdb (con't)

- *Utility:* **gdb** *executableFilename*

  gdb is a UNIX debugger. The named executable file is loaded into the debugger and a user prompt is displayed. To obtain information on the various **gdb** commands, enter **help** at the prompt.

- **Entering the Debugger**

  Once a program has been compiled correctly, you may invoke gdb, with the name of the executable file as the first argument. gdb presents you with a prompt. You enter help at the prompt to see a list of all the gdb commands:

  sepc92:> gdb reverse

  (gdb) *help*

The Unix Debugger : gdb (con't)

```
(gdb)  break reverse
Breakpoint 1 at 0x1077c: file reverse.c, line 22
(gdb) run
Starting program: reverse

Breakpoint 1, reverse(before=0x11360 "cat", …) at reverse.c:22
22          len = strlen(before)
(gdb) display i
1:   i=0
(gdb) display j
2:   j = -4197908
(gdb) step
23          i = 0;
2:   j = -4197908
1:   i = 0
(gdb) step
24          for (j=len-1; j>=0; j--)
2:   j = -4197908
1:   i = 0
(gdb) step
26                  after [i] = before [j]
2:   j = 2
1:   i = 0
```

The Unix Debugger : gdb (con't)

```
(gdb) continue
Continuing.
reverse("cat") = tac

Breakpoint 1, reverse (before=0x4006e1 "noon", …) at reverse.c:22
22          len = strlen(before);
2: j = -668855928
1: i = 62
(gdb) continue
Continuing.
reverse("noon") = noon

Program exited with code 027.
(gdb) quit

sepc92:>
```