① what do you mean by programing language?

Ans: A programamming language is a formal computer language disigned to communicate instructions to a machine, particularly a computer. Programamming language can be used to create programs to control the behavior of a machine or to express algorithms.

② what is C program?

Ans: C is a high-level and general purpose programamming that is ideal for developing firmware or portable applications. C was developed at Bell Labs by Dennis Ritchie for the Unix Operating System (OS) in the early 1970s.

③ what is function?

Ans: A function is a group of statements that together perform a task. Every C program has at least one function, which is main(), and all the most trivial programs can define additional functions. A code can be divided into separate functions.
program can be

④ what is statement?

Ans: A statement specifies an action to be performed by the program. In other words, statements are the part of your program that actually perform operations. All C statements end with a semicolon.

(25) What is do-while loop in C?

Ans: A do-while loop is almost the same as a while loop execept that the loop boody is guaranteed to exeeceete at leaast once. The code must always be exeeceted first and then the test condition is evaluated. If it is true, the code executes the body of the loop again.

```
do{
  ----
  ----
} while (condition);
```

(26) Difer between while and do-while loop?

Ans: I used to think both arein completly same. But some difference have between both. Given below some difference;

While	do-while
1. A while loop will cheek the condition first before executing the content.	1. The do-while loop exeeceteo the content of the loop once before cheeking the condition of the while.
2. Condition is at the begin of the loop block.	2. Condition is at the end of the loop block.
3. While is an entry control loop.	3. do-while is an exit control loop.
4. There is no semicolon at the end of while statement.	4. There is semicolon at the end of while statement.

5. while (a<10)
{printf ("%d", a);
a++;
}
	5. do {
	printf ("%d", a);
	a++;
	} while (a<10);

Machine Languages, Assembly Languages, and High-level Languages

- Three types of computer languages
  2. Assembly language
     - English-like abbreviations representing elementary computer operations
     - Clearer to humans
     - Incomprehensible to computers
       - Translator programs (assemblers)
         - » Convert to machine language

Machine Languages, Assembly Languages, and High-level Languages

- Three types of computer languages
  3. High-level languages
     - Similar to everyday English, use common mathematical notations
     - Single statements accomplish substantial tasks
       - Assembly language requires many instructions to accomplish simple tasks
     - Translator programs (compilers)
       - Convert to machine language

C History

- Developed between 1969 and 1973
- Developed by: **Dennis Ritchie at Bell Labs**

- General purpose programming language.
- Structured programming language

First C Program

```
/* My first simple C program */
#include <stdio.h>
void main ()
{
    printf ("Welcome to C!\n");
}
```

Header

Comments

All C programs have a main function; they also start at main

Function to print to screen

What to print

End of statement

Braces indicate start and end of main

End of line

Relational Operators

- == equality (x == 3)
- != non equality (y != 0)
- < less than (x < y)
- > greater than (y > 10)
- <= less than equal to (x <= 0)
- >= greater than equal to (x >= y)

!!! a==b vs. a=b !!!

Examples

- A < B
- D= b > c;
- If (D)
  A=b+c;
- Mixing with arithmetic op
  - X+Y >= K/3

4	A
2	B
4	C
3	b
4	c
2	X
1	Y
10	K

Logical Operators

- ! not !(x==0)
- && and (x>=0) && (x<=10)
- || or (x>0) || (x<0)

A	B	A && B	A || B	!A	!B
False	False	False	False	True	True
False	True	False	True	True	False
True	False	False	True	False	True
True	True	True	True	False	False

Examples

- A<B && C>=5
- A+B * 2 < 5 && 4>=A/2 || T-2 < 10
- A < B < C ????
- A<B<C is not the same as
  - (A<B) && (B<C)

Precedence for Arithmetic, Relational, and Logical Operators

Precedence	Operation	Associativity
1	( )	Innermost first
2	++ -- + - ! (type)	Right to left (unary)
3	* / %	Left to right
4	+ -	Left to right
5	< <= > >=	Left to right
6	== !=	Left to right
7	&&	Left to right
8	||	Left to right
9	= += -= *= /= %=	Right to left

Exercise

- Assume that following variables are declared
  a = 5.5 b = 1.5 k = -3
- Are the following true or false
  - a < 10.0 + k
  - a + b >= 6.5
  - k != a-b
  - !(a == 3*b)
  - a<10 && a>5

Selection Statements

- if
- if else
- switch

if statement

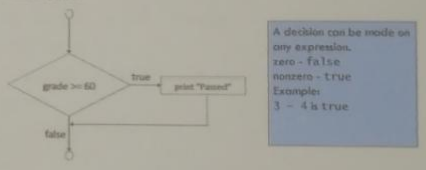
- The if statement allows the program to conditionally execute a program
- General form:
  - `if(Boolean expression)`
    `statement; /* single statement */`
  - `if(Boolean expression) {`
    `/* more than one statement */`
    `statement1;`
    `...`
    `statement n;`
    `}`

if statement

- The if statement allows the program to conditionally execute a program
- The expression can be any valid C expression
  - If the expression is evaluated as true, the statements will be executed
  - Otherwise, the statement is bypassed, and the line of code following the if is executed.
- An expression is true, if it is evaluated to non-zero value
- If it evaluates to zero, it is false
- Commonly, The expression inside the if compares one value with another using relational operator

The `if` Selection Flowchart

- `if` statement is a single-entry/single-exit structure



if statement

- Example:

```
if(10>9)
  printf("true");
if(5>9)
  printf("this will not print");
if(1)
  printf("not zero!");
```

What will be the output?

if else statement

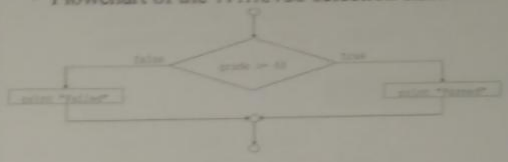
- Using the if else statement, if the expression is **true**, then the statements under if will execute, and the else portion will be skipped
- If the statement is **false**, then the target of the if is bypassed and the target of the else is executed
- Under no circumstance, both statements will execute

if else statement

- General form:
  - `if(Boolean expression)`
    `statement;`
    `else`
    `statement;`
  - `if(Boolean expression) {`
    `statement block`
    `}`
    `else {`
    `statement block`
    `}`

The `if...else` Selection Statement

- Flowchart of the `if...else` selection statement



The `if...else` selection statement

- Example:

```
if ( grade >= 60 )
   printf( "Passed\n");
else
   printf( "Failed\n");
```

if else statement

- What does the following program do?
- Assume that x, y, temp are declared.

```
int x=10, y=20, temp;
 if (x > y)
    temp = x;
 else
    temp = y;
```

Compound Statements

- To make an `if` statement control two or more statements, use a ***compound statement (statement block)***
- A compound statement has the form

  `{ statements }`
- Putting braces around a group of statements forces the compiler to treat it as a single statement.

Compound Statements

- A compound statement is usually put on multiple lines, with one statement per line:

```
{
  line_num = 0;
  page_num++;
}
```

- Each inner statement still ends with a semicolon, but the compound statement itself does not.

Example

```
if(value > 0)
{
    result = 1.0 / value;
    printf("Result = %f\n", result);
}

if(value > 0)
    result = 1.0 / value;
    printf("Result = %f\n", result);
```

Will these two sections of code work differently?

Example

```
if(value > 0)
{
    result = 1.0 / value;
    printf("Result = %f\n", result);
}

if(value > 0)
    result = 1.0 / value;
    printf("Result = %f\n", result);
```

Yes!

Will always execute!

Nested if statement

- When an if statement is the target of another if or else, it is said to be nested within the outer if
- Example:

```
if(count>max) /*outer if*/
   if(error)printf("error, try again.");
   /* nested if*/
```

  - Here, the printf() statement will execute only if count is greater than max and error is nonzero

Nested if statement

- Another example:

```
if(p)
    if(q)
        printf("a and b are true");
else
   Printf("to which statement does this
    else apply?");
```

- An else always associates with the nearest if in the same block, which does not already have and else associated with it

Indentation

```
int a = 750;
if (a>0)
  if (a >= 1000)
    a = 0;
  else
    if (a <500)
      a *= 2;
    else
      a *= 10;
  else
    a += 3;
```

Good

```
int a = 750;
if (a>0)
if (a >= 1000)
a = 0;
else
if (a <500)
a *= 2;
else
a *= 10;
else
a += 3;
```

Not good

If else if ladder

- General form:

```
if(expression)
  statement;
else if (expression)
  statement;
else if (expression)
  statement;
  .
  .
else
  statement;
```

If else if ladder

- The expressions are evaluated from the top downward
- As soon as a true condition is found, the statement associated with it is executed, and the rest of the ladder is bypassed
- If none of the expressions are true, the final else will be executed

Example

- Write a program that takes a number as input, and prints whether the number is positive, negative or 0.

```
#include<stdio.h>
int main(){
int val;
scanf("%d",&val);
if(val < 0)
   printf("Negative value");
else if(val == 0)
   printf("A value of zero");
else
   printf("positive value");
return 0;}
```

Switch statement

- switch statement is the C's multiple selection statement
- It is used to select one of several alternative paths in program
- A expression is matched against a list of constants
- When a match is found, the statement associated with that match is executed

Switch statement

- General form:

```
switch(int expression) {
  case constant1 :
     statement(s);
     break;
  case constant2 :
     statement(s); break;
  /* you can have any number of case
  statements */
  default : /* Optional */
     statement(s); }
```

Example

```
#include <stdio.h>

void main()
{
   int gender;
   printf("Enter your gender
    (male=1, female=2): ");
   scanf("%d",&gender);
```

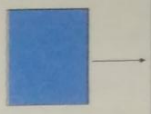
```
switch(gender) {
   case 1:
        printf("You are male\n");
        break;
   case 2:
      printf("you are female\n");
      break;
   default:
        printf(" invalid input\n");
        break;}
}
```

Thank you

Lecture 7

CSE1101: Structured Programming Language

Problem: Conversion table degrees → radians



radians = degrees * PI / 180;

Degrees	to Radians
0	0.000000
10	0.174533
20	0.349066
30	0.523599
…	
340	5.934120
350	6.108653
360	6.283186

Sequential Solution

```
degrees = ???
radians = degrees*PI/180;
printf("%d %f \n", degrees, radians);
```

Not a good solution

```
#include <stdio.h>
#define PI 3.141593

int main(void)
{
  int degrees=0;
  double radians;

  printf("Degrees to Radians \n");

  degrees = 0;
  radians = degrees*PI/180;
  printf("%d %f \n", degrees, radians);

  degrees = 10;
  radians = degrees*PI/180;
  printf("%d %f \n", degrees, radians);

  degrees = 20;
  radians = degrees*PI/180;
  printf("%d %f \n", degrees, radians);
  …
  degrees = 360;
  radians = degrees*PI/180;
  printf("%d %f \n", degrees, radians);
}
```

Loop (Repetition) Structures

- Loops in programs allow us to repeat one or more statements

Loop (Repetition) Structures

- for statement
- while statement
- do while statement
- Two new statements used with loops
  - break and continue

`for` statement

- General form:
  - for(initialization; condition-test; increment or decrement)
    statement;
  - for(initialization; condition-test; increment or decrement)
    {
    statement;
    statement;
    …
    }

Three parts

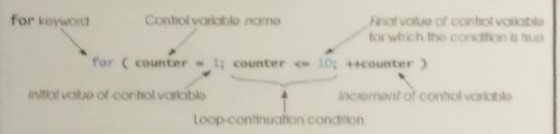
Three parts of a for loop:

- Initialization:
  - Give a initial value to the variable that controls the loop
  - This variable is referred to as loop-control-variable
  - Initialization is executed only once.

Three parts

- Conditional test:
  - this portion tests the loop-control variable against a target value
  - If the test evaluates true, the loop repeats.
  - If it is false, the loop stops, and program execution goes to the next line of code that follows the loop
  - Executed at the start of the loop, each time the loop is repeated
- Increment or decrement:
  - Increase or decrease the loop-control-variable by certain amount
  - Executed at the bottom of the loop

for statement



for Statement

- Format when using for loops

```
for( initialization; conditionTest; increment )
   statement
```

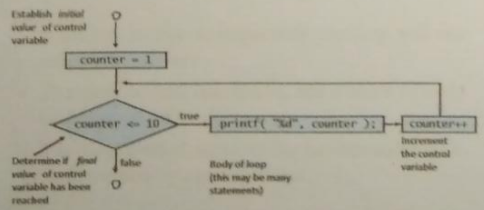
- Example:

```
for(counter = 1; counter <= 10; counter++ )
   printf( "%d\n", counter );
```

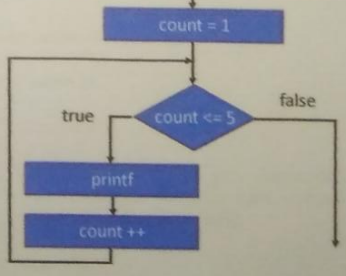
  – Prints the integers from one to ten

No semicolon (;) after last expression

Flow Chart of the Example for Loop



for(count=1; count<=5; count++) printf("count=%d\n", count);



For Example

Use for loop to print the numbers 1 to 5 on the screen

```
#include <stdio.h>

int main()
{
   int num;

   for (num=1; num<6; num = num+1)
       printf("%d",num);

return 0;
}
```

Example

- What will be the output of the following program, also show how values of variables change in the memory.

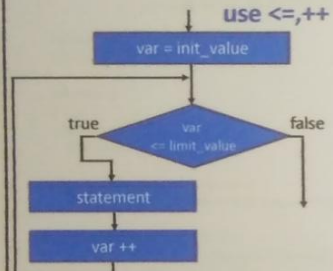
```
int sum1, sum2, k;
sum1 = 0;
sum2 = 0;
for( k = 1; k < 5; k++) {
    if( k % 2 == 0)
      sum1 =sum1 + k;
    else
      sum2 = sum2 + k;
}
printf("sum1 is %d\n", sum1);
printf("sum2 is %d\n", sum2);
```

0 2 6	sum1
0 1 4	sum2
1 2 3 4 5	k

```
sum1 is 6
sum2 is 4
```

Ascending for

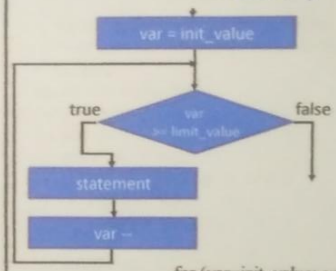
use <=,++




```
for (var=init_value; var <=limit_value;  var++)
    statement;
```

Descending for

use >=,--




```
for (var=init_value; var >=limit_value; var--)
statement;
```

For Example

- If the test is false to begin with, the loop will not execute even once
- To repeat several statements, use a block of code as the target of the loop
- The loop-control variable can be incremented or decremented by more than one.

For Example

```
#include <stdio.h>

int main()
{
   int num;

   /*this loop will not execute*/
   for (num=6; num<6; num = num+1)
       printf("%d",num);

return 0;
}
```

For Example

Compute the product and sum of the numbers from 1 to 5

```
#include <stdio.h>
int main()
{
   int num, sum=0, prod=1;
   for (num=1; num<6; num=num+1){
        sum = sum + num;
        prod = prod * num;
   }
   printf("%d %d", prod, sum);
return 0;
}
```

Examples

```
int sum =0;
for( int i=1 ; i<7 ; i+=2)
      sum = sum + i;

int fact =1;
for(int n=5;n>1;n--)
      fact = fact * n;
```

? 1 3 5 7	i
0 1 4 9	sum
5	n
1	fact

For Example

```
#include <stdio.h>
#define PI 3.141593

int main()
{
  int degrees;
  double radians;

  printf("Degrees to Radians \n");
  for (degrees=0; degrees<=360; degrees+=10)
  {
    radians = degrees*PI/180;
    printf("%d %lf \n", degrees, radians);
  }
  return 0;
}
```

For loop variations

- There does not need to be a loop-control-variable, the conditional expression may use some other means of stopping the loop
- Example:

```
#include<stdio.h>
int main(){
   int i;
   char ch = 'a';
   for( i=0; ch!='q' ; i++){
      printf("%d\n",i);
      scanf("%c",&ch);
   }
   return 0;
}
```

For loop variations

- It is possible to leave an expression in a loop empty
- Example:

```
#include<stdio.h>
int main(){
   int i;
   printf("enter an integer");
   scanf("%d",&i);
   for( ; i ; i--)
      printf("%d ",i);
   return 0;
}
```

For loop variations

- The target of for loop may be empty.
- Example:

```
#include<stdio.h>
int main(){
   char ch;
   for(scanf("%c",&ch);ch!='q';scanf("%c",&ch));
   printf("Found the q");
   return 0;
}
```

For loop variations

- It is possible to create a loop that never stops. It is usually called **infinite loop.**
- Use for construct:

```
for( ; ; )
{

}
```

For loop variations

- It is valid for the loop-control variable to be altered outside the increment section
- Example:

```
#include<stdio.h>
int main(){
   int i;
   for(i=0; i<10;  ){
      printf("%d ",i);
      i++;
   }
   return 0;
}
```

Thank you

Lecture 8

CSE1101: Structured Programming Language

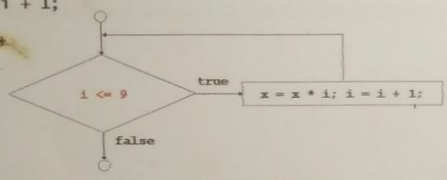
`while` statement

- General form:
  - while(*expression*)
    statement;
  - while(*expression*) {
    statement;
    statement;
    ...
    }

The `while` Control Structure

Example:

```
x = 1; i = 2;
while (i <= 9) {
   x = x * i;
   i = i + 1;
}
```



Example

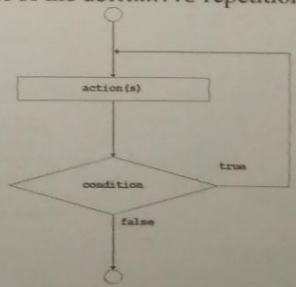
```
#include <stdio.h>
#define PI 3.141593

int main(void)
{
  int degrees=0;
  double radians;

  printf("Degrees to Radians \n");
  while (degrees <= 360)
  {
    radians = degrees*PI/180;
    printf("%d %lf \n", degrees, radians);
    degrees += 10;
  }
return 0;
}
```

`do while`

- General form:
  - do
    statement;
    while(expression);
  - do {
    statement1;
    statement2;
    } while(expression);
- note - the expression is tested *after* the statement(s) are executed, so statements are executed ***at least once.***

The `do...while` Repetition Statement

- Flowchart of the `do...while` repetition statement



Example

```
#include <stdio.h>
#define PI 3.141593

int main(void)
{
 int degrees=0;
 double radians;

 printf("Degrees to Radians \n");
 do
 {
  radians = degrees*PI/180;
  printf("%d %lf \n",degrees,radians);
  degrees += 10;
 } while (degrees <= 360);
 return 0;
}
```

break statement

- **break;**
  - terminates loop
  - execution continues with the first statement following the loop
- Example:

```
#include<stdio.h>
int main(){
  int i;
for (i=1; i<=100; i++) {
    printf("%d ",i);
    if (i == 10)
        break;   /*exit the loop*/
}
return 0;
}
```

break statement

- **break;**
  - The break statement can be used with all three of C's loops
  - The break statement is commonly used in loops in which a special condition can cause immediate termination
  - A break will cause an exit from only the innermost loop

break statement

A break will cause an exit from only the innermost loop

```
#include<stdio.h>
int main(){
  int i, j;
    for(i=0;i<5;i++){
        for(j=0;j<100;j++){
            printf("%d", j);
          if(j==5) break;
          }
     }
      return 0;
}
```

continue statement

- continue;
  - forces next iteration of the loop, skipping any remaining statements in the loop
- Example:

```
#include<stdio.h>
int main(){
  int x;
  for(x=0; x<100; x++){
      continue;
       printf("%d ",x); /*This line never executed*/
  }
  return 0;
  }
```

Example: what will be the output

```
#include<stdio.h>
int main()
{
 int a, b, c;
 a=5;
 while(a > 2) {
  for (b = a ; b < 2 * a ; b++ ) {
     c = a + b;
     if (c < 8) continue;
     if (c > 11) break;
     printf( "a = %d  b = %d   c = %d \n", a, b, c);
  } /* end of for-loop */
  a--;
 } /* end of while loop */
}
```

a = 5 b = 5 c = 10
a = 5 b = 6 c = 11
a = 4 b = 4 c = 8
a = 4 b = 5 c = 9
a = 4 b = 6 c = 10
a = 4 b = 7 c = 11
a = 3 b = 5 c = 8

Nested if....Else Statements

✓ The if..else statement can be contained in another if or else statement.

```
if (test condition1)
{
      if (test condition2)
            statement-1;
      else
            statement-2;
}
else
      statement-3;

statement-x;
```

Nested Loops

- ✓ Similar to nested if statements, loops can be nested as well
- ✓ That is, the body of a loop can contain another loop
- ✓ Any of C's loops can be nested within any other loop
- ✓ Each time through the outer loop, the inner loop goes through its full set of iterations

Exercise

- What is the output of the following program?

```
for (i=1; i<=5; i++) {
      for (j=1; j<=4; j++){
          printf("*");
      }
      printf("\n");
}
```

Output

```
****
****
****
****
****
```

Exercise

- What is the output of the following program?

```
for (i=1; i<=5; i++) {
      for (j=1; j<=i; j++){
          printf("*");
      }
      printf("\n");
}
```

Output

```
*
**
***
****
*****
```

Example: nested loops to generate the following output

```
i=1 *
i=2 + +
i=3 * * *
i=4 + + + +
i=5 * * * * *
```

```
int i, j;
for(i=1; i <= 5; i++){
   printf("i=%d ", i);
   for(j=1; j <= i; j++) {
      if (i % 2 == 0)
         printf("+ ");
      else
         printf("* ");
   }
   printf("\n");
}
```

Modify the following program to produce the output

```
for (i=A; i<=B; i++) {
      for (j=C; j<=D; j++) {
          printf("*");
      }
      printf("\n");
}
```

Output

```
*****
****
***
**
*
```

Exercise

- Write a program using loop statements to produce the output.

Output

```
*
**
***
****
*****
```

goto statement

- A goto statement provides an unconditional jump from the 'goto' to a labeled statement in the same function.
- Use of goto statement is discouraged in any programming language
  - it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify
- The goto statement can perform a jump within a function. It can not jump between functions.

goto statement

- goto statement works with a label
- A label is a valid identifier name followed by a colon
- label can be set anywhere in the C program above or below the goto statement
- General form:

```
goto label;
.
.
.
label: statement;
```

goto statement

- Example:

```
#include<stdio.h>
int main()
{
   goto mylabel;
   printf("This will not print");
   mylabel:
    printf("This will print.");
   return 0;
}
```

Thank you

Lecture 9

CSE1101: Structured Programming Language

Data Type

- C has five basic data types:
  - Whole number (int)
  - Floating point number (float)
  - Double precision floating point number (double)
  - Character data (char)
  - Valueless (void)

Type Modifier

- Basic types can be modified using C's type modifiers:
  - signed
  - unsigned
  - long
  - Short

Signed and Unsigned Integers

- The leftmost bit of an integer is known as sign bit
- The sign bit of a ***signed*** integer is 0 if the number is positive or zero, 1 if it's negative.
- The largest 16-bit integer has the binary representation 0111111111111111, which has the value 32,767 ($2^{15} - 1$).
- The largest 32-bit integer is
  01111111111111111111111111111111
  which has the value 2,147,483,647 ($2^{31} - 1$).

Signed and Unsigned Integers

- An integer with no sign bit (the leftmost bit is considered part of the number's magnitude) is said to be ***unsigned***.
- The largest 16-bit unsigned integer is 65,535 ($2^{16} - 1$).
- The largest 32-bit unsigned integer is 4,294,967,295 ($2^{32} - 1$).

Signed and Unsigned Integers

- By default, integer variables are signed in C
  - the leftmost bit is reserved for the sign
- To tell the compiler that a variable has no sign bit, declare it to be `unsigned`.

Integer Types

- The `int` type is usually 32 bits, but may be 16 bits on older CPUs.
- ***long*** integers may have more bits than ordinary integers;
- ***short*** integers may have fewer bits.
- The specifiers `long` and `short`, as well as `signed` and `unsigned`, can be combined with `int` to form integer types.

Integer Types

- Only six combinations produce different types:

```
short int      unsigned short int
int            unsigned int
long int       unsigned long int
```

- The order of the specifiers doesn't matter.
- Also, the word `int` can be dropped (`long int` can be abbreviated to just `long`).

Integer Types

- The range of values represented by each of the six integer types **varies** from one machine to another.
- However, the C standard requires that `short int`, `int`, and `long int` must each cover a certain minimum range of values.
- Also, `int` must not be shorter than `short int`, and `long int` must not be shorter than `int`.

Integer Types

- Typical ranges of values for the integer types on a 16-bit machine:

Type	*Smallest Value*	*Largest Value*
`short int`	–32,768	32,767
`unsigned short int`	0	65,535
`int`	–32,768	32,767
`unsigned int`	0	65,535
`long int`	–2,147,483,648	2,147,483,647
`unsigned long int`	0	4,294,967,295

Integer Types

- Typical ranges on a 32-bit machine:

Type	*Smallest Value*	*Largest Value*
`short int`	–32,768	32,767
`unsigned short int`	0	65,535
`int`	–2,147,483,648	2,147,483,647
`unsigned int`	0	4,294,967,295
`long int`	–2,147,483,648	2,147,483,647
`unsigned long int`	0	4,294,967,295

Integer Types

- Typical ranges on a 64-bit machine:

Type	*Smallest Value*	*Largest Value*
`short int`	–32,768	32,767
`unsigned short int`	0	65,535
`int`	–2,147,483,648	2,147,483,647
`unsigned int`	0	4,294,967,295
`long int`	-2^{63}	$2^{63}-1$
`unsigned long int`	0	$2^{64}-1$

Integer Types – long long

- Two additional standard integer types, `long long int` and `unsigned long long int`.
- Both `long long` types are required to be at least 64 bits wide.
- The range of `long long int` values is typically -2^{63} (–9,223,372,036,854,775,808) to $2^{63} - 1$ (9,223,372,036,854,775,807).
- The range of `unsigned long long int` values is usually 0 to $2^{64} - 1$ (18,446,744,073,709,551,615).

Constants

- ***Constants*** is a value that can not be altered by the program.
  - For example: the number 100 is a constant
- Constants are also called *literals*
- C allows integer constants to be written in decimal (base 10), octal (base 8), or hexadecimal (base 16).

Octal and Hexadecimal Numbers

- Octal numbers use only the digits 0 through 7.
- Each position in an octal number represents a power of 8.
  - The octal number 237 represents the decimal number $2 \times 8^2 + 3 \times 8^1 + 7 \times 8^0 = 128 + 24 + 7 = 159$.

Octal and Hexadecimal Numbers

- A hexadecimal (or hex) number is written using the digits 0 through 9 plus the letters A through F, which stand for 10 through 15, respectively.
  - The hex number 1AF has the decimal value $1 \times 16^2 + 10 \times 16^1 + 15 \times 16^0 = 256 + 160 + 15 = 431$.

Integer Constants

- ***Decimal*** constants contain digits between 0 and 9, but must not begin with a zero:
  `15  255  32767`
- ***Octal*** constants contain only digits between 0 and 7, and must begin with a zero:
  `017  0377  077777`
- ***Hexadecimal*** constants contain digits between 0 and 9 and letters between `a` and `f`, and always begin with `0x`:
  `0xf  0xff  0x7fff`
- The letters in a hexadecimal constant may be either upper or lower case:
  `0xff  0xfF  0xFf  0xFF  0Xff  0XfF  0XFf  0XFF`

Integer Overflow

- When arithmetic operations are performed on integers, it's possible that the result will be too large to represent.
- For example, when an arithmetic operation is performed on two `int` values, the result must be able to be represented as an `int`.
- If the result can't be represented as an `int` (because it requires too many bits), we say that ***overflow*** has occurred.

Integer Overflow

- The behavior when integer overflow occurs depends on whether the operands were signed or unsigned.
  - When overflow occurs during an operation on *signed* integers, the program's behavior is undefined.
  - When overflow occurs during an operation on *unsigned* integers, the result *is* defined: we get the correct answer modulo 2^n, where n is the number of bits used to store the result.

Reading and Writing Integers

- Reading and writing unsigned, short, and long integers requires new conversion specifiers.
- When reading or writing an ***unsigned*** integer, use the letter `u`, `o`, or `x` instead of `d` in the conversion specification.

```
unsigned int u;

scanf("%u", &u);  /* reads  u in base 10 */
printf("%u", u);  /* writes u in base 10 */
scanf("%o", &u);  /* reads  u in base  8 */
printf("%o", u);  /* writes u in base  8 */
scanf("%x", &u);  /* reads  u in base 16 */
printf("%x", u);  /* writes u in base 16 */
```

Reading and Writing Integers

- When reading or writing a ***short*** integer, put the letter `h` in front of `d`, `o`, `u`, or `x`:

```
short s;

scanf("%hd", &s);
printf("%hd", s);
```

- When reading or writing a ***long*** integer, put the letter `l` ("ell," not "one") in front of `d`, `o`, `u`, or `x`.
- When reading or writing a ***long long*** integer, put the letters `ll` in front of `d`, `o`, `u`, or `x`.

Representation of negative number

- Negative numbers are usually represented using two's complement approach
- All bits in the number (except the sign flag) are reversed and 1 is added to the number
- Finally, the sign flag is set to 1

Floating Types

- C provides three ***floating types***, corresponding to different floating-point formats:
  - `Float` Single-precision floating-point
  - `double` Double-precision floating-point
  - `long double` Extended-precision floating-point

Floating Types

- `float` is suitable when the amount of precision isn't critical.
- `double` provides enough precision for most programs.
- `long double` is rarely used.

- The C standard doesn't state how much precision the `float`, `double`, and `long double` types provide, since that depends on how numbers are stored.
- Most modern computers follow the specifications in IEEE Standard 754 (also known as IEC 60559).

The IEEE Floating-Point Standard

- IEEE Standard 754 was developed by the Institute of Electrical and Electronics Engineers.
- It has two primary formats for floating-point numbers: single precision (32 bits) and double precision (64 bits).
- Numbers are stored in a form of scientific notation, with each number having a ***sign***, an ***exponent***, and a ***fraction***.
- In single-precision format, the exponent is 8 bits long, while the fraction occupies 23 bits. The maximum value is approximately 3.40×10^{38}, with a precision of about 6 decimal digits.

Floating Types

- Characteristics of `float` and `double` when implemented according to the IEEE standard:

Type	*Storage size*	*Smallest Positive Value*	*Largest Value*	*Precision*
`float`	4 byte	1.17549×10^{-38}	3.40282×10^{38}	6 digits
`double`	8 byte	2.22507×10^{-308}	1.79769×10^{308}	15 digits

- On computers that don't follow the IEEE standard, this table won't be valid.

Floating Constants

- Floating constants can be written in a variety of ways.
- Valid ways of writing the number 57.0:

```
57.0  57.  57.0e0  57E0  5.7e1  5.7e+1
.57e2  570.e-1
```

- A floating constant must contain a decimal point and/or an exponent; the exponent indicates the power of 10 by which the number is to be scaled.
- If an exponent is present, it must be preceded by the letter `E` (or `e`). An optional `+` or `-` sign may appear after the `E` (or `e`).

Floating Constants

- **By default**, floating constants are stored as double-precision numbers.
- To indicate that only single precision is desired, put the **letter `F`** (or `f`) at the end of the constant (for example, `57.0F`).
- To indicate that a constant should be stored in `long double` format, put the letter `L` (or `l`) at the end (`57.0L`).

Reading and Writing Floating-Point Numbers

- The conversion specifications `%e`, `%f`, and `%g` are used for reading and writing single-precision floating-point numbers.
- When reading a value of type **`double`**, put the **letter `l`** in front of `e`, `f`, or `g`:

```
double d;
scanf("%lf", &d);
```

- When reading or writing a value of type **`long double`**, put the **letter `L`** in front of `e`, `f`, or `g`.

Character Types

- The only remaining basic type is **`char`**, the character type.

Character Sets

- Today's most popular character set is ***ASCII*** (American Standard Code for Information Interchange), a 7-bit code capable of representing 128 characters.
- ASCII is often extended to a 256-character code known as ***Latin-1*** that provides the characters necessary for Western European and many African languages.

Character Sets

- A variable of type `char` can be assigned any single character:

```
char ch;

ch = 'a';   /* lower-case a */
ch = 'A';   /* upper-case A */
ch = '0';   /* zero         */
ch = ' ';   /* space        */
```

- Notice that character constants are enclosed in **single quotes**, not double quotes.

Operations on Characters

- Working with characters in C is simple, because of one fact: *C treats characters as small integers.*
- In ASCII, character codes range from 0000000 to 1111111, which we can think of as the integers from 0 to 127.
- The character `'a'` has the value 97, `'A'` has the value 65, `'0'` has the value 48, and `' '` has the value 32.

Operations on Characters

- When a character appears in a computation, C uses its integer value.
- Consider the following examples, which assume the ASCII character set:

```
char ch;
int i;

i = 'a';        /* i is now 97   */
ch = 65;        /* ch is now 'A' */
ch = ch + 1;    /* ch is now 'B' */
ch++;           /* ch is now 'C' */
```

Operations on Characters

- Characters can be compared, just as numbers can.
- An `if` statement that converts a lower-case letter to upper case:

```
if (ch >= 'a' && ch <= 'z')
  ch = ch - 'a' + 'A';
```

- Comparisons such as `ch>='a'` are done using the integer values of the characters involved.

Operations on Characters

- The fact that characters have the same properties as numbers has advantages.
- For example, it is easy to write a `for` statement whose control variable steps through all the upper-case letters:

```
for (ch = 'A'; ch <= 'Z'; ch++) …
```

Signed and Unsigned Characters

- The `char` type—like the integer types—exists in both signed and unsigned versions.
- Signed characters normally have values between –128 and 127. Unsigned characters have values between 0 and 255.
- C allows the use of the words `signed` and `unsigned` to modify `char`:

```
signed char sch;
unsigned char uch;
```

Reading and Writing Characters Using `scanf` and `printf`

- The `%c` conversion specification allows `scanf` and `printf` to read and write single characters:

```
char ch;

scanf("%c", &ch);  /* reads one character */
printf("%c", ch);  /* writes one character */
```

- `scanf` doesn't skip white-space characters.

Reading and Writing Characters Using `scanf` and `printf`

- A loop that reads and ignores all remaining characters in the current input line:

```
do {
  scanf("%c", &ch);
} while (ch != '\n');
```

- When `scanf` is called the next time, it will read the first character on the next input line.

Reading and Writing Characters Using `getchar` and `putchar`

- For single-character input and output, `getchar` and `putchar` are an alternative to `scanf` and `printf`.
- `putchar` writes a character:

```
putchar(ch);
```

- Each time `getchar` is called, it reads one character, which it returns:

```
ch = getchar();
```

- Like `scanf`, `getchar` doesn't skip white-space characters as it reads.

Reading and Writing Characters Using `getchar` and `putchar`

- Using `getchar` and `putchar` (rather than `scanf` and `printf`) saves execution time.
  - `getchar` and `putchar` are much simpler than `scanf` and `printf`, which are designed to read and write many kinds of data in a variety of formats.

Reading and Writing Characters Using `getchar` and `putchar`

- Consider the `scanf` loop that we used to skip the rest of an input line:

```
do {
  scanf("%c", &ch);
} while (ch != '\n');
```

- Rewriting this loop using `getchar` gives us the following:

```
do {
  ch = getchar();
} while (ch != '\n');
```

Reading and Writing Characters Using `getchar` and `putchar`

- Moving the call of `getchar` into the controlling expression allows us to condense the loop:

```
while ((ch = getchar()) != '\n');
```

- The `ch` variable isn't even needed; we can just compare the return value of `getchar` with the new-line character:

```
while (getchar() != '\n');
```

Reading and Writing Characters Using `getchar` and `putchar`

- `getchar` is useful in loops that skip characters as well as loops that search for characters.
- A statement that uses `getchar` to skip an indefinite number of blank characters:

```
while ((ch = getchar()) == ' ');
```

- When the loop terminates, `ch` will contain the first nonblank character that `getchar` encountered.

Reading and Writing Characters Using `getchar` and `putchar`

- Be careful when mixing `getchar` and `scanf`.
- `scanf` has a tendency to leave behind characters that it has "peeked" at but not read, including the new-line character.

```
printf("Enter an integer: ");
scanf("%d", &i);
printf("Enter a command: ");
command = getchar();
```

`scanf` will leave behind any characters that weren't consumed during the reading of `i`, including (but not limited to) the new-line character.

- `getchar` will fetch the first leftover character.

Program: Determining the Length of a Message

- The `length.c` program displays the length of a message entered by the user:

```
Enter a message: Brevity is the soul of wit.
Your message was 27 character(s) long.
```

- The length includes spaces and punctuation, but not the new-line character at the end of the message.
- We could use either `scanf` or `getchar` to read characters;
- `length2.c` is a shorter program that eliminates the variable used to store the character read by `getchar`.

```
/* Determines the length of a message */

#include <stdio.h>

int main(void)
{
  char ch;
  int len = 0;

  printf("Enter a message: ");
  ch = getchar();
  while (ch != '\n') {
    len++;
    ch = getchar();
  }
  printf("Your message was %d character(s)
  long.\n", len);

  return 0;
}
```

```
/* Determines the length of a message */

#include <stdio.h>

int main(void)
{
  int len = 0;

  printf("Enter a message: ");
  while (getchar() != '\n')
    len++;
  printf("Your message was %d character(s)
  long.\n", len);

  return 0;
}
```

Lecture 4

CSE1101: Structured Programming Language

Variable

- Most programs need to a way to store data temporarily during program execution.
- A **variable** is a named memory location that can hold various values.

Data Type

- Every variable must have a **type**.
- C has five basic data types:
  - Whole number (int)
  - Floating point number (float)
  - Double precision floating point number (double)
  - Character data (char)
  - Valueless (void)

Data Type

- A variable of type `int` (short for *integer*) can store a whole number such as 0, 1, 392, or –2553
  - numbers with no fractional part
  - The largest `int` value is typically 2,147,483,647 but can be as small as 32,767.
  - Integers are usually 16 or 32 bits long.

Data Type

- A variable of type `float` (short for *floating-point*) can store much larger numbers than an `int` variable.
- Also, a `float` variable can store numbers with digits after the decimal point, like 379.125.

Declarations

- Variables must be ***declared*** before they are used.
- Variables can be declared one at a time:
  - General form: `type var-name;`
  - Example:
```
int height;
float profit;
```
- Alternatively, several can be declared at the same time:
  - General form: `type var-name1, var-name2;`
  - Example:
```
int height, length, width;
float profit, loss;
```

Declarations

- When `main` contains declarations, these should precede statements:

```
int main(void)
{
  declarations
  statements
}
```

Variable name

- A variable name in C can consist of the letters of the alphabet, the digits and the underscore
- A digit may not start a variable's name
- C is case sensitive:
  - `Count` and `COUNT` is two different variable name

Assignment

- To assign a value to a variable, put its name to the left of an equal sign
  - put the value you want to give, to the right of the equal sign
- General form of assignment:

```
var-name = value;
```

- Example:

```
int height;
height = 8;
```

The number `8` is said to be a ***constant.***

- Before a variable can be assigned a value—or used in any other way—it must first be declared.

Assignment

- A constant assigned to a `float` variable usually contains a decimal point:

```
profit = 2150.48;
```

- It's best to append the letter `f` to a floating-point constant if it is assigned to a `float` variable:

```
profit = 2150.48f;
```

Failing to include the `f` may cause a warning from the compiler.

Expression

- Once a variable has been assigned a value, it can be used to help compute the value of another variable:

```
height = 8;
length = 12;
width = 10;
volume = height * length * width;
  /* volume is now 960 */
```

- The right side of an assignment can be a formula (or ***expression***, in C terminology) involving constants, variables, and operators.

Printing the Value of a Variable

- `printf` can be used to display the current value of a variable.
- To write the message

`Height: h`

where ***h*** is the current value of the `height` variable, we'd use the following call of `printf`:

```
printf("Height: %d\n", height);
```

- `%d` is a placeholder indicating where the value of `height` is to be filled in.

Format specifier

- %d is known as a **format specifier**
  - It informs printf() that a different type item is to be displayed
- `%d` works only for `int` variables
- To print a `float` variable, use `%f` instead.
  - By default, `%f` displays a number with six digits after the decimal point.
- To print a `double` variable, use `%lf`
- To print a `char` variable, use `%c`

Printing the Value of a Variable

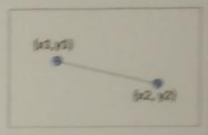
- There's no limit to the number of variables that can be printed by a single call of `printf`:

```
printf("Height: %d  Length: %d\n", height, length);
```

Problem Solving Methodology

1. **Read the problem clearly**
2. **Understand the input/output information**
3. **Work the problem by hand, give example**
4. **Develop a solution** (Algorithm Development) **and Convert it to a program** (C program)
5. **Test the solution with a variety of data**

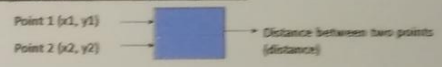
Example 1



1. Problem statement

Compute the straight line distance between two points in a plane

2. Input/output description



Example 1 (cont'd)

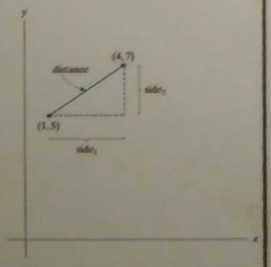
3. Hand example

$side1 = 4 - 1 = 3$
$side2 = 7 - 5 = 2$

$distance = \sqrt{side1^2 + side2^2}$

$distance = \sqrt{3^2 + 2^2}$

$distance = \sqrt{13} = 3.61$



Example 1 (cont'd)

4. Algorithm development and coding

a. Generalize the hand solution and list/outline the necessary operations step-by-step
   1) Give specific values for point1 (x1, y1) and point2 (x2, y2)
   2) Compute side1=x2-x1 and side2=y2-y1
   3) Compute $distance = \sqrt{side1^2 + side2^2}$
   4) Print distance

b. Convert the above outlined solution to a program using any language you want (see next slide for C imp.)

Format specifier

- %d is known as a **format specifier**
  - It informs printf() that a different type item is to be displayed
- `%d` works only for `int` variables
- To print a `float` variable, use `%f` instead.
  - By default, `%f` displays a number with six digits after the decimal point.
- To print a `double` variable, use `%lf`
- To print a `char` variable, use `%c`

Printing the Value of a Variable

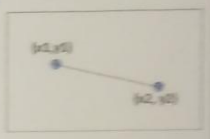
- There's no limit to the number of variables that can be printed by a single call of `printf`:

```
printf("Height: %d  Length: %d\n", height, length);
```

Problem Solving Methodology

1. **Read the problem clearly**
2. **Understand the input/output information**
3. **Work the problem by hand, give example**
4. **Develop a solution** (Algorithm Development)
   and Convert it to a program (C program)
5. **Test the solution with a variety of data**

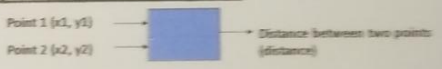
Example 1



1. Problem statement

Compute the straight line distance between two points in a plane

2. Input/output description



Example 1 (cont'd)

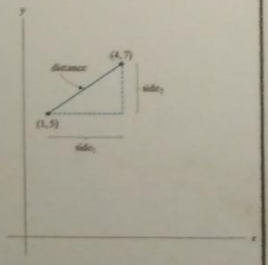
3. Hand example

side1 = 4 - 1 = 3
side2 = 7 - 5 = 2

$distance = \sqrt{side1^2 + side2^2}$

$distance = \sqrt{3^2 + 2^2}$

$distance = \sqrt{13} = 3.61$



Example 1 (cont'd)

4. Algorithm development and coding

a. Generalize the hand solution and list/outline the necessary operations step-by-step
   1) Give specific values for point1 (x1, y1) and point2 (x2, y2)
   2) Compute side1=x2-x1 and side2=y2-y1
   3) Compute $distance = \sqrt{side1^2 + side2^2}$
   4) Print distance

b. Convert the above outlined solution to a program using any language you want (see next slide for C imp.)

Identifiers

- Names for variables, functions, macros, and other entities are called *identifiers*.
- An identifier may contain
  - letters,
  - digits, and
  - underscores,
  - **but must begin with a letter or underscore:**

Keywords

- This 32 keywords may not be used as identifiers

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

Fig: C's reserved keywords.

Comments

- A comment is a note to yourself
- All comments are ignored by the compiler
- Comments are used to document the meaning and purpose of your source code, so that you can remember later how it functions and how to use it.

Comments

- In C, the start of a comment is signaled by /*
- A comment is ended by */
- Example:

```
/* This is a comment*/
```

- It can also extend over several lines.
- Example:

```
/*
This is a longer comment.
This is the second line of comment
*/
```

Comments

- Another style of comment is called a single line comment
- It starts with // and stops at the end of the line
- Example:

```
// This is a comment
```

- In C, you can not have one comment within another comment

Arithmetic operators

- Expression: a statement which is a combination of operators and operands
- C defines 5 arithmetic operators:

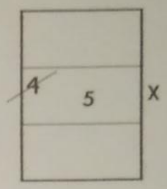
Addition	+	sum = num1 + num2;
Subtraction	-	age = 2007 – my_birth_year;
Multiplication	*	area = side1 * side2;
Division	/	avg = total / number;
Modulus	%	lastdigit = num % 10;

Arithmetic operators

- Modulus:
  - Modulus returns remainder of division between two *integers*
  - Example 5%2 returns a value of 1
  - % may be used with integer types only
- Binary vs. Unary operators
  - All the previous operators are binary
  - - is an unary operator, e.g., a = -3 * -4

Arithmetic Operators (cont'd)

- Note that 'id = exp' means assign the result of exp to id, so
- X=X+1 means
  - first perform X+1 and
  - Assign the result to X
- Suppose X is 4, and
- We execute X=X+1



Integer division vs Real division

- Division between two integers results in an integer.
- The result is truncated, not rounded
- Example:
  int A=5/3; → A will have the value of 1
  int B=3/6; → B will have the value of 0
- To have floating point values:
  double A=5.0/3; → A will have the value of 1.666
  double B=3.0/6.0; → B will have the value of 0.5

Precedence of Arithmetic Operators

Precedence	Operator	Associativity
1	Parentheses: ( )	Innermost first
2	Unary operators: + - (type)	Right to left
3	Binary operators: * / %	Left to right
4	Binary operators: + -	Left to right
5	assign =	Right to left

Mixed operations:

int a=4+6/3*2; → a=? a= 4+2*2 = 4+4 = 8

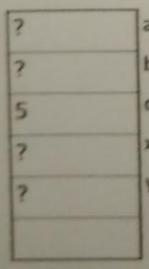
int b=(4+6)/3*2; → b=? b= 10/3*2 = 3*2= 6

Exercise: Arithmetic operations

- Show the memory snapshot after the following operations by hand

```
int a, b, c=5;
double x, y;
a = c * 2.5;
b = a % c * 2 - 1;
x = (5 + c) * 2.5;
y = x - (-3 * a) / 2;
```

Write a C program and print out the values of a, b, c, x, y and compare them with the ones that you determined by hand.



a = 12 b = 3 c= 5 x = 25.0000 y = 43.0000

Exercise: Arithmetic operations

- Show how C will perform the following statements and what will be the final output?

```
int a =  6, b = -3, c =  2;
c = a - b * (a + c * 2) + a / 2 * b;
printf("Value of c = %d \n", c);
```

Step-by-step show how C will perform the operations

c = 6 - -3 * (6 + 2 * 2) + 6 / 2 * -3;
c = 6 - -3 * (6 + 4) + 3 * -3
c = 6 - -3 *10 + -9
c = 6 - -30 + -9
c = 36 + -9
c = 27

- output:

  Value of c = 27

Lecture 13

CSE1101: Structured Programming Language

String

- In C, a string is defines as a null terminated character array
- null: '\0' character, ASCII value is 0
- The string size should be declared such that it can hold the null character at the end, along with other characters
- A string constant is null terminated automatically.

String

- Example:
  - `char pet[5] = {'l', 'a', 'm', 'b','\0'};`
  - `char pet[5] = "lamb";`
- A **string literal** is a sequence of characters enclosed within double quotes:

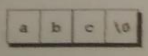
```
"When you come to a fork in the road, take it."
```

How String Literals Are Stored

- When a C compiler encounters a string literal of length *n* in a program, it sets aside *n* + 1 bytes of memory for the string.
- This memory will contain the characters in the string, plus one extra character—the ***null character***—to mark the end of the string.
- The null character is a byte whose bits are all zero, so it's represented by the '\0'

How String Literals Are Stored

- The string literal `"abc"` is stored as an array of four characters:



- The string `""` is stored as a single null character:

String Literals versus Character Constants

- A string literal containing a single character isn't the same as a character constant.
  - `"a"` is a string literal
  - `'a'` is a character constant

String Variables

- If a string variable needs to hold 80 characters, it must be declared to have length 81:

```
char str[80+1];
```

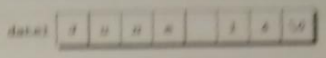
- Adding 1 to the desired length allows room for the null character at the end of the string.
- Failing to do so may cause unpredictable results when the program is executed.

Initializing a String Variable

- A string variable can be initialized at the same time it's declared:

```
char date1[8] = "June 14";
```

- The compiler will automatically add a null character so that `date1` can be used as a string:

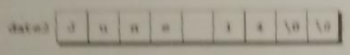


Initializing a String Variable

- If the initializer is too short to fill the string variable, the compiler adds extra null characters:

```
char date2[9] = "June 14";
```

Appearance of `date2`:



Initializing a String Variable

- The declaration of a string variable may omit its length, in which case the compiler computes it:

```
char date4[] = "June 14";
```

- The compiler sets aside eight characters for `date4`, enough to store the characters in `"June 14"` plus a null character.
- Omitting the length of a string variable is especially useful if the initializer is long, since computing the length by hand is error-prone.

Reading and Writing Strings

- Writing a string is easy using either `printf` or `puts`.
- To read a string in a single step, we can use either `scanf` or `gets`.
- As an alternative, we can read strings one character at a time.

Writing Strings Using `printf` and `puts`

- The `%s` conversion specification allows `printf` to write a string:

```
char str[] = "Are we having fun yet?";
printf("%s\n", str);
```

The output will be

```
Are we having fun yet?
```

- `printf` writes the characters in a string one by one until it encounters a null character.

String functions

strcpy(s1, s2);

Copies string s2 into string s1.

strcat(s1, s2);

Concatenates string s2 onto the end of string s1.

strlen(s1);

Returns the length of string s1.

strcmp(s1, s2);

Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.

strchr(s1, ch);

Returns a pointer to the first occurrence of character ch in string s1.

strstr(s1, s2);

Returns a pointer to the first occurrence of string s2 in string s1.

Example

```
#include <stdio.h>
#include <string.h>
 int main () {
char str1[12] = "Hello", str2[12] = "World", str3[12];
int len ;
strcpy(str3, str1);  /* copy str1 into str3 */
printf("strcpy( str3, str1) : %s\n", str3 );
strcat( str1, str2); /*concatenates str1 and str2 */
printf("strcat( str1, str2): %s\n", str1 );
len = strlen(str1);/* total length of str1 after
   concatenation */
printf("strlen(str1) : %d\n", len );
 return 0;
}
```

Example

```
#include <stdio.h>
#include <string.h>
int main () {
char str1[15], str2[15];
int ret;
scanf("%s %s",str1, str2);
ret = strcmp(str1, str2);
if(ret < 0)
  printf("str1 is less than str2");
else if(ret > 0)
  printf("str2 is less than str1");
else printf("str1 is equal to str2");
 return 0;
 }
```

Array of strings

- Arrays of strings are also called string tables
- Example: `char names[10][10];`
  - This specifies a table than can contain 10 strings, each up to 10 characters long, including the null character
  - To read a string within this table, specify the leftmost index, example: `gets(names[0]);`
  - Similarly, to output the first string: `printf("%s",names[0]);`

Array of strings

- You can initialize a string table as you would any other type of array;
- Example:

```
Char name[3][10]= {"bob","tim","harry"};
```

Thank you

Lecture 11

CSE1101: Structured Programming Language

Local Variables

- A variable declared in the body of a function is said to be ***local*** to the function:

```
int sum_digits(int n)
{
  int sum = 0;   /* local variable */

  while (n > 0) {
    sum += n % 10;
    n /= 10;
  }

  return sum;
}
```

Local Variables

- Default properties of local variables:
  - ***Automatic storage duration.*** Storage is "automatically" allocated when the enclosing function is called and deallocated when the function returns.
  - ***Block scope.*** A local variable is visible from its point of declaration to the end of the enclosing function body.

Local Variables

- Since C99 doesn't require variable declarations to come at the beginning of a function, it's possible for a local variable to have a very small scope:

```
void f(void)
{
  ...
  int i;  ┐
  ...     ├ scope of i
}         ┘
```

Parameters

- Parameters have the same properties—automatic storage duration and block scope—as local variables.
- Each parameter is initialized automatically when a function is called (by being assigned the value of the corresponding argument).

Global Variables

- Passing arguments is one way to transmit information to a function.
- Functions can also communicate through ***global variables***—variables that are declared outside the body of any function.

Global Variables

- Properties of global variables:
  - Static storage duration
  - File scope
- Having *file scope* means that an global variable is visible from its point of declaration to the end of the enclosing file.

Pros and Cons of Global Variables

- Global variables are convenient when many functions must share a variable or when a few functions share a large number of variables.
- In most cases, it's better for functions to communicate through parameters rather than by sharing variables:
  - If we change an global variable during program maintenance (by altering its type, say), we'll need to check every function in the same file to see how the change affects it.
  - If an global variable is assigned an incorrect value, it may be difficult to identify the guilty function.
  - Functions that rely on global variables are hard to reuse in other programs.

Pros and Cons of global Variables

- Don't use the same global variable for different purposes in different functions.
- Suppose that several functions need a variable named `i` to control a `for` statement.
- Instead of declaring `i` in each function that uses it, some programmers declare it just once at the top of the program.
- This practice is misleading; someone reading the program later may think that the uses of `i` are related, when in fact they're not.

Pros and Cons of global Variables

- Make sure that global variables have meaningful names.

Pros and Cons of global Variables

- Making variables global when they should be local can lead to some rather frustrating bugs.
- Code that is supposed to display a 10 × 10 arrangement of asterisks:

```
int i;

void print_one_row(void)
{
  for (i = 1; i <= 10; i++)
    printf("*");
}

void print_all_rows(void)
{
  for (i = 1; i <= 10; i++) {
    print_one_row();
    printf("\n");
  }
}
```

- Instead of printing 10 rows, `print_all_rows` prints only one.

Blocks

- Compound statements are of the form

  { *statements* }
- C allows compound statements to contain declarations as well as statements:

  { *declarations statements* }
- This kind of compound statement is also called a *block.*

Blocks

- By default, the storage duration of a variable declared in a block is automatic: storage for the variable is allocated when the block is entered and deallocated when the block is exited.
- The variable has block scope; it can't be referenced outside the block.

Blocks

- The body of a function is a block.
- Blocks are also useful inside a function body when we need variables for temporary use.
- C99 and C++ allows variables to be declared anywhere within a block.

Scope of a variable

- Scope refers to the portion of the program in which
  - It is valid to reference the variable
  - The variable is visible or accessible

Scope

- In a C program, the same identifier may have several different meanings.
- C's scope rules enable the programmer (and the compiler) to determine which meaning is relevant at a given point in the program.
- The most important scope rule: When a declaration inside a block names an identifier that's already visible, the new declaration temporarily "hides" the old one, and the identifier takes on a new meaning.
- At the end of the block, the identifier regains its old meaning.

Scope

- In the example on the next slide, the identifier `i` has four different meanings:
  - In Declaration 1, `i` is a variable with static storage duration and file scope.
  - In Declaration 2, `i` is a parameter with block scope.
  - In Declaration 3, `i` is an automatic variable with block scope.
  - In Declaration 4, `i` is also automatic and has block scope.
- C's scope rules allow us to determine the meaning of `i` each time it's used (indicated by arrows).

```
int i;                  /* Declaration 1 */

void f(int i)           /* Declaration 2 */
{
  i = 1;
}

void g(void)
{
  int i = 2;            /* Declaration 3 */

  if (i > 0) {
    int i;              /* Declaration 4 */

    i = 3;
  }

  i = 4;
}

void h(void)
{
  i = 5;
}
```

Scope

- *Local* scope
  - a local variable is defined within a function or a block and can be accessed only within the function or block that defines it
- *Global* scope
  - a global variable is defined outside the **main** function and can be accessed by any function within the program file.

Global vs Local V

```
#include <stdio.h>
int z = 2;
void function1()
{ int a = 4;
  printf("Z = %d\n",z);
  z = z+a;
}
int main()
{ int a = 3;
  z = z + a;
  function1();
  printf("Z = %d\n",z);
  z = z+a;
  return 0;
}
```

Output

Z = 5
Z = 9

Scope rules

- Identifiers (i.e. variables etc) are accessible **only** within the block in which they are declared. Example:

```
{
    int a = 2;                  /* outer block a */
    printf("%d\n", a);              /* 2 is printed */
    {
        int a = 5;              /* inner block a */
        printf("%d\n", a);          /* 5 is printed */
    }
    printf("%d\n", a);              /* 2 is printed */
}
```

- A variable that is declared in an outer block is available in the inner block unless it is redeclared, in which case the outer block declaration is temporarily "masked".

Thank you