LEHIGH UNIVERSITY

Introduction to C Programming

Alexander B. Pacheco

Research Computing
Lehigh University

Outline

Introduction

What is the C Language?

- A general-purpose, procedural, imperative computer programming language.
- Developed in 1972 by Dennis M. Ritchie at the Bell Telephone Laboratories to develop the UNIX operating system.
- The UNIX operating system, the C compiler, and essentially all UNIX applications programs have been written in C.
- C is the most widely used computer language.
  - Easy to learn
  - Structured language
  - Produces efficient programs
  - Handles low-level activities
  - Can be compiled on a variety of computer plaforms
- Most of the state-of-the-art softwares have been implemented using C.
- Today's most popular Linux OS and RBDMS MySQL have been written in C.

What do you need to learn C?

1. C Compiler
   - What is a Compiler?
     - A compiler is a computer program (or set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language, often having a binary form known as object code).
   - How does a compiler do?
     - Translate C source code into a binary executable
   - List of Common Compilers:
     - GCC GNU Project (Free, available on most *NIX systems)
     - Intel Compiler
     - NVIDIA HPC SDK (formerly Portland Group (PGI) Compiler)
     - Microsoft Visual Studio
2. Text Editor
   - Emacs
   - VI/VIM
   - Notepad++ (avoid Notepad if you will eventually use a *NIX system)
   - Integrated Development Environment: Eclipse, XCode, Visual Studio, etc

Basics

Program Structure

A C Program consists of the following parts

- Preprocessor Commands
- Functions
- Variables
- Statements & Expressions
- Comments

A Simple Hello World Code

```c
#include <stdio.h>

int main ()
{
  /* My First C Code */
  printf("Hello World!\n");
  return 0;
}
```

Compile and execute the code

```
dyn100077:Exercise apacheco$ gcc hello.c
dyn100077:Exercise apacheco$ ./a.out
Hello World!
```

My First C Code

```c
#include <stdio.h>

int main ()
{
  /* My First C Code */
  printf("Hello World!\n");
  return 0;
}
```

- `#include <stdio.h>` is a preprocessor command.

  It tells a C compiler to include stdio.h file before going to actual compilation.
- `int main()` is the main function where program execution begins.
- `/* ... */` is a comment and ignored by the compiler.
- `printf(...)` is function that prints `Hello World!` to the screen.
- `return 0;` terminates main() function and returns the value 0.

- C is a case sensitive programming language i.e. program is not the same as Program or PROGRAM.
- Each individual statement must end with a semicolon.
- Whitespace i.e. tabs or spaces is insignificant except whitespace within a character string.
- All C statments are free format i.e. no specified layout or column assignment as in FORTRAN77.

```c
#include <stdio.h>
int main () {  /* My First C Code */  printf("Hello World!\n");  return 0;}
```

  will produce the exact same result as the code on the previous slide.

- In C everything within `/*and */` is a comment. Comments can span multiple lines.

```c
/* this is single line comment */
/* This
is a
multiline comment */
```

- Always use proper comments in your code. Your code will most likely be handed to someone long after you are gone.
- Comments are completely ignored by compiler (test/debug code)

Valid Character Set in C language

Alphabets	ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
Digits	0123456789

Special Characters															
,	_	{	<	’	(	^	;	$	/	*	+	[	#	?	
.	&	}	>	"	)	!	:	%	|	\	-	]	~		

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

- White space Characters: blank space, new line, horizontal tab, carriage return and form feed

Data Types

LEHIGH UNIVERSITY

Basic Types: There are five basic data types

1. int - integer: a whole number.
2. float - floating point value: ie a number with a fractional part.
3. double - a double-precision floating point value.
4. char - a single character.
5. void - valueless special purpose type.

Derived Types: These include

1. Pointers
2. Arrays
3. Structures
4. Union
5. Function

- The array and structure types are referred to collectively as the aggregate types.
- The type of a function specifies the type of the function's return value.

Basic Data Types: Integer

LEHIGH UNIVERSITY

Type	Storage size (in bytes)	Value range
char	1	-128 to 127 or 0 to 255
unsigned char	1	0 to 255
signed char	1	-128 to 127
int	2 or 4	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4	0 to 65,535 or 0 to 4,294,967,295
short	2	-32,768 to 32,767
unsigned short	2	0 to 65,535
long	4	-2,147,483,648 to 2,147,483,647
unsigned long	4	0 to 4,294,967,295

- To get the exact size of a type or a variable on a particular platform, you can use the sizeof operator.
- The expressions `sizeof(type)` yields the storage size of the object or type in bytes.

Basic Data Types: Floating-Point & void

LEHIGH UNIVERSITY

Type	Storage size	Value range	Precision (decimal places)
float	4 bytes	1.2E-38 to 3.4E38	6
double	8 bytes	2.3E-308 to 1.7E308	15
long double	10 bytes	3.4E-4932 to 1.1E4932	19

Situation	Description
function returns as void	function with no return value
function arguments as void	function with no parameter
pointers to void	address of an object without type

Variables

- Variables are memory location in computer's memory to store data.
- To indicate the memory location, each variable should be given a unique name called identifier.
- Variable names are just the symbolic representation of a memory location.
- Rules for variable names:
  1. Composed of letters (both uppercase and lowercase letters), digits and underscore '_' only.
  2. The first letter of a variable should be either a letter or an underscore.
  3. There is no rule for the length of a variable name.
     - Most likely your code will be used by someone else, so variable names should be meaningful and short as possible.

```
int num;
float circle_area;
double _volume;
```

- In C programming, you have to declare variable before using it in the program.

Declaring Variable or Variable Definition

LEHIGH UNIVERSITY

- A variable definition means to tell the compiler where and how much to create the storage for the variable.
- A variable definition specifies a data type and contains a list of one or more variables of that type as follows:

```
type variable_list;
```

- `type` must be a valid C data type or any user-defined object, etc., and `variable_list` may consist of one or more identifier names separated by commas.
- Variables can be initialized (assigned an initial value) in their declaration.

```
type variable_name = value;
```

```
int    i, j, k;
char   c, ch;
float  f, salary;
double d;
int d = 3, f = 5;           // definition and initializing d and f.
byte z = 22;                // definition and initializes z.
char x = 'x';               // the variable x has the value 'x'.
```

Constants & Literals

LEHIGH UNIVERSITY

The constants refer to fixed values that the program may not alter during its execution. These fixed values are also called literals.

Integer Constants

```
85      /* decimal */
0213    /* octal */
0x4b    /* hexadecimal */
30      /* int */
30u     /* unsigned int */
30l     /* long */
30ul    /* unsigned long */
```

Character Constants

```
'a'     /* character 'a' */
'Z'     /* character 'Z' */
\?      /*? character */
\\      /*\ character */
\n      /*Newline */
\r      /*Carriage return */
\t      /*Horizontal tab */
```

Floating Point Constants

```
3.1416
314159E-5     /* 3.14159 */
2.1E+5        /* 2.1x10^5 */
3.7E-2        /* 0.037 */
0.5E7         /* 5.0x10^6 */
-2.8E-2       /* -0.028 */
```

String Constants

```
"hello, world"          /* normal string */
"c programming \
language"               /* multi-line string */
```

How to define Constants

- Constants can be defined in two ways
  1. Using the **#define** preprocessor (defining a macro)
  2. Using the **const** keyword (new standard borrowed from C++)

```
#include <stdio.h>

/* define LENGTH using the macro */
#define LENGTH 5

int main()
{
 /*define WIDTH using const */
  const int WIDTH = 3;
  const char NEWLINE = '\n';
  int area = LENGTH * WIDTH;

  printf("value of area : %d", area);
  printf("%c", NEWLINE);
  return 0;
}
```

Input and Output

- C or any programming language in general needs to be interactive i.e. write something back and optionally read data to be useful.
- Similar to Unix, C treats all devices as files.

Standard File	File Pointer	Device
Standard Input	stdin	Keyboard
Standard Output	stdout	Screen
Standard Error	stderr	Screen

- C Programming language provides three functions to read/write from standard input/output

	Unformatted		Formatted
Input	getchar	gets	scanf
Output	putchar	puts	printf

The getchar() & putchar() functions

- The `int getchar(void)` function reads the next available character from the screen and returns it as an integer.

  This function reads only single character at a time.

- The `int putchar(int c)` function puts the passed character on the screen and returns the same character.

  This function puts only single character at a time.

The gets() & puts() functions

- The `char *gets(char *s)` function reads a line from stdin into the buffer pointed to by s until either a terminating newline or EOF.
- The `int puts(const char *s)` function writes the string s and a trailing newline to stdout.

```c
#include <stdio.h>
int main( )
{
   int c;

   printf( "Enter a value :");
   c = getchar( );

   printf( "\nYou entered: ");
   putchar( c );

   return 0;
}
```

```c
#include <stdio.h>
int main( )
{
   char str[100];

   printf( "Enter a value :");
   gets( str );

   printf( "\nYou entered: ");
   puts( str );

   return 0;
}
```

Formatted I/O

- The `int scanf(const char *format, ...)` function reads input from the standard input stream stdin and scans that input according to format provided.
- The `int printf(const char *format, ...)` function writes output to the standard output stream stdout and produces output according to a format provided (optional).

```
#include <stdio.h>

int main ()
{
  /* My Second C Code */
  char name[100];
  printf("Enter your name:");
  scanf("%s",&name);
  printf("Hello %s\n",name);
  return 0;
}
```

- In this program, the user is asked a input and value is stored in variable `name`.
- Note the '`&`' sign before `name`.
- `&name` denotes the address of `name` and value is stored in that address.

Common Format Specifier

- The format specifier: `%[flags][width][.precision][length]specifier`

flag	meaning
-	left justify
+	always display sign
0	pad with leading zeros

Specifier	Output	Example
%f	decimal float	3.456
%7.5f	decimal float, 7 digit width and 5 digit precision	3.45600
%d	integer	5
%05d	integer, 5 digits pad with zeros	00101
%s	string of characters	"Hello World!"
%e	scientific notation for decimal float	2.71828e+5
%c	character	
\n	insert new line	
\t	insert tab	

```c
/* printf example showing different specifier usage */
#include <stdio.h>
int main() {
  printf ("Characters: %c %c \n", 'a', 65);
  printf ("Decimals: %d %0qa4d\n", 2014, 65);
  printf ("\t floats: %7.5f \t%f \t%e \n", 3.1416, 3.1416, 3.1416);
  printf ("%s \n", "hello world");
  return 0;
}
```

```
alexanders-mbp:Example apacheco$ gcc -o print print.c
alexanders-mbp:Example apacheco$ ./print
Characters: a A
Decimals: 2014 0065
         floats: 3.14160        3.141600        3.141600e+00
hello world
```

Operators

- Arithmetic

Operator	Meaning
+	addition or unary plus
-	subtraction or unary minus
*	multiplication
/	division
%	remainder after division(modulo division)
++	increase integer value by one
--	decrease integer value by one

- Assignment Operator

Operator	Example	Same as
=	a=b	a=b
+=	a+=b	a=a+b
-=	a-=b	a=a-b
*=	a*=b	a=a*b
/=	a/=b	a=a/b
%=	a%=b	a=a%b

Increment/Decrement Operator

- There are two types of increment/decrement operators
  1. Suffix or Postfix: e.g. i++ or j--
     a=i++ means set a to i and then increment i by 1
  2. Prefix: ++i or --j
     a=++i means increment i by 1 and then set a to i
- Consider the following example
  If i = 1 and j = 2, then
  ++i + j++ = 4
  and not 5 since j is incremented after the operation is complete

```c
#include<stdio.h>

int main () {
  int i=1,j=2;
  int a, b;
  int k=1,l=2;

  a=++k ;
  b=l++ ;

  printf("++i + j++: %d\n", ++i + j++ );
  printf("a=++i: %d, b=j++: %d, i:%d, j:%d\n", a, b
         , k, l);
  printf("a(=++i) + b(=j++): %d\n", a + b);

  return 0;
```

```
alexanders-mbp:Example apacheco$ make increment
cc     increment.c   -o increment
alexanders-mbp:Example apacheco$ ./increment
++i + j++: 4
a=++i: 2, b=j++: 2, i:2, j:3
a(=++i) + b(=j++): 4
```

Relational Operators

- Relational operators checks relationship between two operands.
- If the relation is true, it returns value 1 and if the relation is false, it returns value 0.
- Relational operators are used in decision making and loops in C programming.

Operator	Meaning	Example
==	Equal to	5==3 returns false (0)
>	Greater than	5>3 returns true (1)
<	Less than	5<3 returns false (0)
!=	Not equal to	5!=3 returns true(1)
>=	Greater than or equal to	5>=3 returns true (1)
<=	Less than or equal to	5<=3 return false (0)

Logical & Conditional Operators

LEHIGH UNIVERSITY

- Logical operators are used to combine expressions containing relation operators.
- In C, there are 3 logical operators

Operator	Meaning	Example
&&	Logial AND	If c=5 and d=2 then,((c==5) && (d>5)) returns false.
||	Logical OR	If c=5 and d=2 then, ((c==5) || (d>5)) returns true.
!	Logical NOT	If c=5 then, !(c==5) returns false.

- Conditional Operator: Conditional operators are used in decision making in C programming, i.e, executes different statements according to test condition whether it is either true or false.

```
conditional_expression?expression1:expression2
```

- If the test condition is true, expression1 is returned and if false expression2 is returned.

```
d=(c>0)?10:-10;
```

If c is greater than 0, value of d will be 10 but, if c is less than 0, value of d will be -10.

Other Operators

- Bitwise Operators: works on bits and perform bit-by-bit operation

Truth Table				
p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

- Misc Operators

Operator	Description
sizeof()	Returns the size of an variable.
&	Returns the address of an variable.
*	Pointer to a variable.
? :	Conditional Expression

Operator Precedance

Operator	Description	Associativity
++, --	Suffix Increment/Decrement	→
++, --	Prefix Increment/Decrement	←
+, -	Unary plus and minus	
!, ~	Logical NOT and Bitwise NOT	
*	Indirection (dereference)	
&	Address of	
sizeof	Size-of	
*, /, %	Multiplication, division, modulo	→
+, -	Addition, Subtraction	
«, »	Bitwise left and right shift	
<, <=	Relational Operators	
>, >=		
==, !=		
&	Bitwise AND	
^	Bitwise XOR	
|	Bitwise OR	
&&	Logical AND	
||	Logical OR	
?:	Ternary Conditional	←
=	Simple Assignment	
+=, -=	Assignment by sum and difference	
*=, /=, %=	Assignment by product, quotient and remainder	
«=, »=	Assignment by bitwise left and right shift	
&=, ^=, |=	Assignment by logical AND, XOR and OR	
,	Comma Operator	→

Control Flow

Control Flow

- Conditional Statements (decision making/selection)
  - if $\cdots$ else if $\cdots$ else
  - switch
- Loops
  - for
  - while
  - do while

if statement

- An if statement consists of a boolean expression followed by one or more statements.

```
if(expression)
{
   /* statement(s) will execute if the boolean expression is true */
}
```

- If the boolean expression evaluates to true, then the block of code inside the if statement will be executed.
- If boolean expression evaluates to false, then the first set of code after the end of the if statement(after the closing curly brace) will be executed.

if · · · else statement

- An if statement can be followed by an optional else statement, which executes when the boolean expression is false.

```
if(expression)
{
   /* statement(s) will execute if the boolean expression is true */
}
else
{
  /* statement(s) will execute if the boolean expression is false */
}
```

- If the boolean expression evaluates to true, then the if block of code will be executed, otherwise else block of code will be executed.

if · · · else if · · · else statement

- An if statement can be followed by an optional else if· · · else statement,
- very useful to test various conditions using single if· · · else if statement.
- When using if , else if , else statements there are few points to keep in mind:
  - An if can have zero or one else's and it must come after any else if's.
  - An if can have zero to many else if's and they must come before the else.
  - Once an else if succeeds, none of the remaining else if's or else's will be tested.

```
if(expression 1)
{
   /* Executes when the boolean expression 1 is true */
}
else if( expression 2)
{
   /* Executes when the boolean expression 2 is true */
}
else if( expression 3)
{
   /* Executes when the boolean expression 3 is true */
}
else
{
   /* executes when the none of the above condition is true */
}
```

```c
#include <stdio.h>

int main ()
{
   /* local variable definition */
   int a = 100;

   /* check the boolean condition */
   if( a < 20 )
   {
       /* if condition is true then print the following */
       printf("a is less than 20\n" );
   }
   else
   {
       /* if condition is false then print the following */
       printf("a is not less than 20\n" );
   }
   printf("value of a is : %d\n", a);

   return 0;
}
```

Nested if· · · else statement

- You can use one if or else if statement inside another if or else if statement(s) i.e. nested if· · · else statement/s

```
if( expression 1)
{
   /* Executes when the boolean expression 1 is true */
   if( expression 2)
   {
      /* Executes when the boolean expression 2 is true */
   }
}
```

```c
#include <stdio.h>

int main ()
{
   /* local variable definition */
   int a = 100;
   int b = 200;

   /* check the boolean condition */
   if( a == 100 )
   {
       /* if condition is true then check the following */
       if( b == 200 )
       {
          /* if condition is true then print the following */
          printf("Value of a is 100 and b is 200\n" );
       }
   }
   printf("Exact value of a is : %d\n", a );
   printf("Exact value of b is : %d\n", b );

   return 0;
}
```

switch statement

- A switch statement allows a variable to be tested for equality against a list of values.
- Each value is called a case, and the variable being switched on is checked for each switch case.

```
switch(expression){
    case constant-expression  :
       statement(s);
       break; /* optional */
    case constant-expression  :
       statement(s);
       break; /* optional */

    /* you can have any number of case statements */
    default : /* Optional */
       statement(s);
}
```

- The expression used in a switch statement must have an integral type

  (or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type).

switch statement

- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The constant-expression for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.
- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached.
- A switch statement can have an optional default case, which must appear at the end of the switch.
- The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

switch statement

```c
#include <stdio.h>

int main ()
{
   /* local variable definition */
   char grade;
   printf("Enter your grade:\n");
   scanf("%c", &grade);

   switch(grade)
   {
   case 'A' :
      printf("Excellent!\n" );
      break;
   case 'B' :
   case 'C' :
      printf("Well done\n" );
      break;
   case 'D' :
      printf("You passed\n" );
      break;
   case 'F' :
      printf("Better try again\n" );
      break;
   default :
      printf("Invalid grade\n" );
   }
   printf("Your grade is  %c\n", grade );

   return 0;
}
```

Nested Conditional Statements

- Conditional statements can be nested as they do not overlap:

```
if( expression 1) {
  if(expression 2) {
    /* Executes when the boolean expression 2 is true */
    /* nested switch statement */
    switch(expression){
    case constant-expression :
      statement(s);
      break; /* optional */
    case constant-expression :
      statement(s);
      break; /* optional */
      /* you can have any number of case statements */
    default : /* Optional */
      statement(s);
    }
  }
 }
```

for loop

- A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.
  - The init step is executed first and only once.
  - the condition is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute, the loop exits.
  - the increment statement executes after the loop body.
  - The loop continues until the condition becomes false

```
for ( init; condition; increment )
{
   statement(s);
}
```

while and do· · · while loops

- while loops are similar to for loops
- A while loop continues executing the code block as long as the condition in the while holds.

```
while(condition)
{
   statement(s);
}
```

- do· · · while loop is guaranteed to execute at least one time.

```
do
{
   statement(s);

}while( condition );
```

Simple loops using for, while, do while

```c
#include <stdio.h>
int main ()
{
  int i;
  /* for loop execution */
  for(i = 0; i < 5; i++ ) {
    printf("for loop i= %d\n", i);
  }
  i=0;
  /* while loop execution */
  while( i < 5 ) {
    printf("while loop i: %d\n", i);
    i+=1;
  }
  i=1;
  /* do-while loop execution */
  do {
    printf("do while loop i: %d\n", i);
    i=i+1;
  }while( i < 0 );

  return 0;
}
```

Nested loops in C

- All loops can be nested as long as they do not overlap

```c
/* nested for loops*/
for (init; condition; increment) {
  for (init; condition; increment) {
    statement(s);
  }
  statement(s);
 }
/* nested while loops*/
while (condition) {
  while (condition) {
    statement(s);
  }
  statement(s);
 }
```

```c
/* nested do while loops*/
do {
  statement(s);
  do {
    statement(s);
  } while ( condition );
 } while ( condition );
/* mixed type loops*/
while (condition) {
  for (init; condition; increment) {
    statement(s);
    do {
      statement(s);
    } while ( condition );
  }
  statement(s);
 }
```

```c
#include <stdio.h>

int main () {
  int i, j, k, n=2;
  printf("i j k\n");
  /* Nested for loops */
  for (i=0; i<n; ++i)
    for (j=0; j<n; j++)
      for (k=0; k<n; ++k)
  printf("%d %d %d\n", i,j,k);
  return 0;
}
```

Loop Control Statement

- Loop control statements change execution from its normal sequence.
  - **break:** Terminates the loop or switch statement
  - **continue:** Causes the loop to skip the remainder of its body for the current iteration
  - **goto:** Transfers control to the labeled statement. Use is not advised

```c
#include <stdio.h>

int main ()
{
  /* local variable definition */
  int a = 10;

  /* while loop execution */
  while( a < 20 )
    {
      printf("value of a: %d\n", a);
      a++;
      if( a > 15)
  {
    /* terminate the loop using break statement */
          break;
  }
    }

  return 0;
}
```

```c
#include <stdio.h>

int main ()
{
  /* local variable definition */
  int a = 10;

  /* do loop execution */
  do
    {
      if( a == 15)
  {
    /* skip the iteration */
    a = a + 1;
    continue;
  }
      printf("value of a: %d\n", a);
      a++;

    }while( a < 20 );

  return 0;
}
```

Functions

Functions

- A function is a group of statements that together perform a task.
- Every C program has at least one function, which is main()
- Functions receive either a fixed or variable amount of arguments.
- Functions can only return one value, or return no value (void).
- In C, arguments are **passed by value** to functions
- How to return value? - **Pointers**
- Functions are defined using the following syntax:

```
return_type function_name( parameter list )
{
  body of the function
}
```

- A function **declaration** tells the compiler about a function's name, return type, and parameters.
- A function **definition** provides the actual body of the function.

Function Definition

- **Return Type:** Function's return type is the data type of the value the function returns. When there is no return value, return void.
- **Function Name:** This is the actual name of the function.
- **Parameter:** The parameter list refers to the type, order, and number of the parameters of a function. A function may contain no parameters.
- **Function Body:** The function body contains a collection of statements that define the function behavior.

```
/* function returning the max between two numbers */
int max(int i, int j)
{
  /* local variable declaration */
  int result;

  if (i > j)
    result = i;
  else
    result = j;

  return result;
}
```

Example of using a Function

```c
#include <stdio.h>

/* function declaration */
int max(int i, int j);

int main() {

  /* local variable definition */
  int i = 100, j = 200, maxval;

  /* calling a function to get max value */
  maxval = max(a, b);

  printf( "Max value is : %d\n", maxval );
  return 0;

}


/* function returning the max between two numbers */
int max(int i, int j)
{
  /* local variable declaration */
  int result;

  if (i > j)
    result = i;
  else
    result = j;

  return result;
}
```

Scope Rules: Local & Global Variables

LEHIGH UNIVERSITY

- A scope is a region of the program where a defined variable can have its existence and beyond that variable can not be accessed.
- **Local Variables:** declared inside a function or block.

  can be used only by statements that are inside that function or block of code.

  Local variables are not known to functions outside their own.
- **Global Variables:** defined outside of a function, usually on top of the program.

  will hold their value throughout the lifetime of your program and,

  they can be accessed inside any of the functions defined for the program.
- A program can have same name for local and global variables but value of local variable inside a function will take preference.

Scope Rules: Local & Global Variables

```c
#include <stdio.h>

/* global variable declaration */
int a = 20;

int main ()
{
  /* local variable declaration in main function */
  int a = 10;
  int b = 20;
  int c = 0;

  printf ("value of a in main() = %d\n",  a);
  c = sum( a, b);
  printf ("value of c in main() = %d\n",  c);

  return 0;
}

/* function to add two integers */
int sum(int a, int b)
{
  printf ("value of a in sum() = %d\n",  a);
  printf ("value of b in sum() = %d\n",  b);

  return a + b;
}
```

Scope Rules: Local & Global Variables

```
value of a in main() = 10
value of a in sum() = 10
value of b in sum() = 20
value of c in main() = 30
```

Initializing Local & Global Variables

- Local Variables are not initialized by the system, the programmer must initialize it.
- Global variables are automatically initialized by the system depending on the data type

Data Type	Initial Default Value
int	0
char	'\0'
float	0
double	0
pointer	NULL

- *It is a good programming practice to initialize variables properly otherwise, your program may produce unexpected results because uninitialized variables will take some garbage value already available at its memory location.*

Arrays

- Arrays are special variables which can hold more than one value using the same name with an index.
- Declaring Arrays: `type arrayName [ arraySize ];`

```
/* simply define the arrays */
double balance[10];
float atom[1000];
int index[5];
```

- C array starts its index from 0

[0]	[1]	[2]	[3]	[4]
10	15	14	3	7

index[2] (3rd element of the array) has a value 14

- Initialize arrays with values

```
/* initialize the array with values*/
double atmass[4] = {12.0, 1.0, 1.0, 16.0};
double atmass[] = {12.0, 1.0, 1.0, 16.0};
atmass[0] = 12.0
```

- Access array values via index

```
/* access the array values*/
int current_index = index[i];
double current_value=value[current_cell_index];
```

Array Example

```c
#include <stdio.h>

int main ()
{
  int n[ 10 ]; /* n is an array of 10 integers */
  int i,j;

  /* initialize elements of array n to 0 */
  for ( i = 0; i < 10; i++ )
    {
      n[ i ] = i + 100; /* set element at location i to i + 100 */
    }

  /* output each array element's value */
  for (j = 0; j < 10; j++ )
    {
      printf("Element[%d] = %d\n", j, n[j] );
    }

  return 0;
}
```

Accessing C arrays

- C arrays are a sequence of elements with contiguous addresses.
- There is no bounds checking in C.
- Be careful when accessing your arrays
- Compiler will not give you error, you will have *undefined* runtime behavior:

```c
#include <stdio.h>

int main() {

  int index[5]={5, 4, 6, 3, 1};

  int a=3;

  /* undefined behavior */

  printf("%d\n",index[5]);

}
```

Multidimensional Arrays

- General form of multidimensional array

```
type name[size1][size2]...[sizeN];
```

- Declaring 2D and 3D arrays:

```
float array2d[4][5];
double array3d[2][3][4];
```

- Initializing multidimensional arrays

```
int a[3][4] = {{/* 2D array is composed of 1D arrays*/
  {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
  {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
  {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
  };
```

	col0	**col1**	**col2**	**col3**
row0	a[0][0]=0	a[0][1]=1	a[0][2]=2	a[0][3]=3
row1	a[1][0]=4	a[1][1]=5	a[1][2]=6	a[1][3]=7
row2	a[2][0]=8	a[2][1]=9	a[2][2]=10	a[2][3]=11

- C arrays are **row major** order i.e. in memory, the C array appears as

a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[1][0]	a[1][1]	· · ·	a[1][3]	a[2][0]	· · ·	a[2][3]

Example: Arrays

```c
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

int main () {
  /* Program to calculate the sum, min and max of an integer array */
  int i, sum, min, max, n=11  ;
  int a[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

  sum = max = 0.0 ; min = 10.0 ;
  /* Initialize array */

  /* Find sum, min and max */
  for (i = 0 ; i < n ; i++ ) {
    sum += a[i] ;
    if (a[i] > max ) max = a[i];
    if (a[i] < min ) min = a[i];
  }

  printf("The max value is: %d\n", max);
  printf("The min value is: %d\n", min);
  printf("The sum value is: %d\n", sum);
  return 0;

}
```

Strings in C

- Strings in C are a special type of array: array of characters terminated by a null character '\0'.

```
/* define string */
char str[7]={'H', 'E', 'L', 'L', 'O', '!', '\0'};
char str1="HELLO!";
```

- Memory presentation of above defined string in C/C++:

str[]	[0]	[1]	[2]	[3]	[4]	[5]	[6]
	'H'	'E'	'L'	'L'	'O'	'!'	'\0'

- C uses built-in functions to manipulate strings:

```
/* C sample 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. */
```

Strings in C

```c
#include <stdio.h>
#include <string.h>

int main ()
{
  char str1[12] = "Hello";
  char str2[12] = "World";
  char str3[12];
  int  len ;

  /* copy str1 into str3 */
  strcpy(str3, str1);
  printf("strcpy( str3, str1) :  %s\n", str3 );

  /* concatenates str1 and str2 */
  strcat( str1, str2);
  printf("strcat( str1, str2):   %s\n", str1 );

  /* total lenghth of str1 after concatenation */
  len = strlen(str1);
  printf("strlen(str1) :  %d\n", len );

  return 0;
}
```

File Input/Output

Opening & Closing Files

- Opening Files: use the `fopen( )` function to create a new file or to open an existing file, this call will initialize an object of the type FILE

```
FILE *fopen( const char * filename, const char * mode );
```

  - `filename` is string literal, which you will use to name your file and access `mode` can have one of the following values:

Mode	Description
r	Read Only, file pointer is at beginning of file
w	Write Only, file pointer is at beginning of file
a	Append, if file exists, file pointer is at end of file
r+	Read & Write
w+	first truncate the file to zero length if it exists otherwise create the file if it does not exist.
a+	creates file if it does not exist. The reading will start from the beginning but writing can only be appended.

- Closing Files: use the `fclose( )` function.

```
int fclose( FILE *fp );
```

  - The `fclose( )` function returns zero on success, or EOF if there is an error in closing the file.
  - This function actually, flushes any data still pending in the buffer to the file, closes the file, and releases any memory used for the file.
  - The EOF is a constant defined in the header file stdio.h.

Writing Files

- simplest function to write individual characters to a stream:

```
int fputc( int c, FILE *fp );
```

- function `fputc()` writes the character value of the argument 'c' to the output stream referenced by `fp`.
- returns the written character written on success otherwise EOF if there is an error.
- to write a null-terminated string to a stream:

```
int fputs( const char *s, FILE *fp );
```

- function `fputs()` writes the string 's' to the output stream referenced by `fp`.
- returns a non-negative value on success, otherwise EOF is returned in case of any error.
- You can use `int fprintf(FILE *fp,const char *format, ...)` function as well to write a string into a file.

Reading Files

- simplest function to read a single character from a file:

```
int fgetc( FILE * fp );
```

- `getc()` unction reads a character from the input file referenced by `fp`.
- return value is the character read, or in case of any error it returns EOF.
- functions to read a string from a stream:

```
char *fgets( char *buf, int n, FILE *fp );
```

- function `fgets()` reads up to $n - 1$ characters from the input stream referenced by `fp`.
- It copies the read string into the buffer buf, appending a null character to terminate the string.

Example: Writing & Reading a File

LEHIGH UNIVERSITY

```c
#include <stdio.h>

main()
{
  FILE *fp;

  fp = fopen("/tmp/test.txt", "w+");
  fprintf(fp, "This is testing for fprintf...\n
        ");
  fputs("This is testing for fputs...\n", fp);
  fclose(fp);
}
```

```c
#include <stdio.h>

main()
{
  FILE *fp;
  char buff[255];

  fp = fopen("/tmp/test.txt", "r");
  fscanf(fp, "%s", buff);
  printf("1 : %s\n", buff );

  fgets(buff, 255, (FILE*)fp);
  printf("2: %s\n", buff );

  fgets(buff, 255, (FILE*)fp);
  printf("3: %s\n", buff );
  fclose(fp);

  }
```

Preprocessor

C Preprocessor

- The C Preprocessor is not part of the compiler, but is a separate step in the compilation process.
- In simplistic terms, a C Preprocessor is just a text substitution tool and they instruct compiler to do required pre-processing before actual compilation.
- All preprocessor commands begin with a pound symbol (#).
- It must be the first nonblank character, and for readability, a preprocessor directive should begin in first column.

Directive	Description
#define	Substitutes a preprocessor macro
#include	Inserts a particular header from another file
#undef	Undefines a preprocessor macro
#ifdef	Returns true if this macro is defined
#ifndef	Returns true if this macro is not defined
#if	Tests if a compile time condition is true
#else	The alternative for #if
#elif	#else an #if in one statement
#endif	Ends preprocessor conditional
#error	Prints error message on stderr
#pragma	Issues special commands to the compiler, using a standardized method

- replace instances of MAX_ARRAY_LENGTH with 20

```
#define MAX_ARRAY_LENGTH 20
```

- get stdio.h from System Libraries and add the text to the current source file.

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

- get myheader.h from the local directory and add the content to the current source file.

```
#include "myheader.h"
```

- undefine existing FILE_SIZE and define it as 42.

```
#undef FILE_SIZE

#define FILE_SIZE 42
```

- define MESSAGE only if MESSAGE isn't already defined.

```
#ifndef MESSAGE
#define MESSAGE "You wish!"
#endif
```

- process the statements enclosed if DEBUG is defined.

```
#ifdef DEBUG
/* Your debugging statements here */
#endif
```

- This is useful if you pass the -DDEBUG flag to gcc compiler at the time of compilation.

Pointers

Pointers

- Pointers are a very important part of the C programming language.
- They are used in many ways, such as:
  - Array operations (e.g., while parsing strings)
  - Dynamic memory allocation
  - Sending function arguments by reference
  - Generic access to several similar variables
  - Malloc data structures of all kinds, especially trees and linked lists
  - Efficient, by-reference "copies" of arrays and structures, especially as function parameters
- Necessary to understand memory and address $\cdots$ and the C programming language.

What is a Pointer

- A pointer is essentially a **variable** whose value is the address of another variable.
- Since it is a variable, it must be declared before use.
- Pointer "points" to a specific part of the memory.
- How to define pointers?

```
/* type: pointer's base type
var-name: name of the pointer variable.
asterisk *:designate a variable as a pointer */
type *pointer_var_name;
```

- Examples

```
int *i_ptr; /* pointer to an integer */
double *d_ptr; /* pointer to a double */
float *f_ptr; /* pointer to a float */
char *ch_ptr; /* pointer to a character */
int **p_ptr; /* pointer to an integer pointer */
```

Pointer Rules

- There are two prefix unary operators to work with pointers.

```
& /*"address of" operator */

* /*"dereferencing" operator */
```

- Use ampersand "**&**" in front of a variable to access it's address, this can be stored in a pointer variable.
- Use asterisk "*****" in front of a pointer you will access the value at the memory address pointed to (**dereference** the pointer).
- Example

```
int a = 8;
int *p;
/* point p to a */
p = &a;
/* dereference pointer p */
*p = 10;
```

Part of symbol table

var_name	var_address	var_value
a	bff5a400	8
p	bff5a3f6	bff5a400

Pointer to variables and dereference pointers

```c
/* pointer_rules.c */

#include <stdio.h>

int main() {

  int a = 6, b = 10;
  int *p;

  printf("\nInitial values:\n\tthe value of a is %d, value of b is %d\n", a, b);
  printf("the address of a is : %p, address of b is : %p\n", &a, &b);
  p = &a; /* point p to a */
  printf("\nafter \"p = &a\":\n");
  printf("\tthe value of p is %p, value at that address is %d\n", p, *p);
  p = &b; /* point p to b */
  printf("\nafter \"p = &b\":\n");
  printf("\tthe value of p is %p, value at that address is %d\n", p, *p);
  /* dereference pointer p */
  *p = 6, p = &a, *p = 10 ;
  printf("\nafter dereferencing the pointer:\n");
  printf("\tthe value of a is %d, value of b is %d\n", a, b);
  return 0;
}
```

Never dereference an uninitialized pointer!

- In order to dereference the pointer, pointer must have a valid value (address).
- What is the problem for the following code?

```
int *ptr;
*ptr = 3;
```

- Again, you will have **undefined behavior** at runtime, you are operating on unknown memory space.
- Typically error: "Segmentation fault", possible illegal memory operation
- **Always initialize your variables before use!**

var_name	var_address	var_value
ptr	0x22aac0	0xXXXX
	0xXXXX	3

NULL Pointer

- Memory address 0 has special significance, if a pointer contains the null (zero) value, it is assumed to point to nothing, defined as NULL in C.
- Set the pointer to NULL if you do not have exact address to assign to your pointer.
- A pointer that is assigned NULL is called a null pointer.

```
/* set the pointer to NULL 0 */
int *ptr = NULL;
```

- Before using a pointer, ensure that it is not equal to NULL:

```
if (ptr != NULL) {
  /* make use of pointer1 */
  /* ... */
  }
```

Pointers and Functions

- In C, arguments are passed by value to functions: changes of the parameters in functions do **not** change the parameters in the calling functions.
- Take a look at the below example, what are the values of a and b after we called swap(a, b);

```c
/* this is the main calling function */

int main() {

  int a = 2;
  int b = 3;

  printf("Before: a = %d and b = %d\n", a, b );
  swap( a, b );
  printf("After: a = %d and b = %d\n", a, b );

}

/* this is function, pass by value */
void swap(int p1, int p2) {

  int t;

  t = p2, p2 = p1, p1 = t;
  printf("Swap: a (p1) = %d and b(p2) = %d\n", p1, p2 );

}
```

Pointers and Functions

- The values of a and b do not change after calling swap(a,b)
- **Pass by value means the called function's parameter will be a copy of the caller's passed argument**. The value of the caller and called functions will be the same, but the identity (the variable) is different - caller and called function each has its own copy of parameters

```
/* this is function, pass by reference */
void swap_by_reference(int *p1, int *p2) {

  int t;

  t = *p2, *p2 = *p1, *p1 = t;
  printf("Swap: a (p1) = %d and b(p2) = %d\n", *p1, *p2);

}


/* call by-address function */
swap_by_reference( &a, &b );
```

- The most frequent use of pointers in C is for walking efficiently along arrays.
- **Remember, array name is the first element address of the array (it is a constant)**

```c
int *p=NULL; /* define an integer pointer p*/
/* array name represents the address of the 0th element of the array
     */
int a[5]={1,2,3,4,5};
/* for 1d array, below 2 statements are equivalent */
p = &a[0]; /* point p to the 1st array element (a[0])'s address */
p = a; /* point p to the 1st array element (a[0])'s address */
*(p+1); /* access a[1] value */
*(p+i); /* access a[i] value */
p = a+2; /* p is now pointing at a[2] */
p++; /* p is now at a[3] */
p--; /* p is now back at a[2] */
```

- Recall 2D array structure: combination of 1D arrays

  ```c
  int a[2][2]={{1,2},{3,4}};
  ```

- The 2D array contains 2 1D arrays: array a[0] and array a[1]
- a[0] is the address of a[0][0], i.e:
  - a[0] $\Leftrightarrow$ &a[0][0]
  - a[1] $\Leftrightarrow$ &a[1][0]
- **Array a** is then actually an **address array** composed of a[0], a[1], i.e. a $\Leftrightarrow$ &a[0]

Walk through array with pointer

```c
#include <stdio.h>

const int MAX = 3;

int main () {

  int a_i[] = {10, 20, 30};
  double a_f[] = {0.5, 1.5, 2.5};
  int i;
  int *i_ptr;
  double *f_ptr;

  /* let us have array address in pointer */
  i_ptr = a_i;
  f_ptr = a_f;

  /* use the ++ operator to move to next location */
  for (i=0; i<MAX; i++,i_ptr++,f_ptr++ ) {
    printf("adr a_i[%d] = %8p\t", i, i_ptr );
    printf("adr a_f[%d] = %8p\n", i, f_ptr );
    printf("val a_i[%d] = %8d\t", i, *i_ptr );
    printf("val a_f[%d] = %8.2f\n", i, *f_ptr );
  }
  return 0;

}
```

Dynamic memory allocation using pointers

- For situations that the size of an array is unknown, we must use pointers to dynamically manage storage space.
- C provides several functions for memory allocation and management.
- Include <stdlib.h> header file to use these functions.
- Function prototype:

```
/* This function allocates a block of num bytes of memory and
     return
a pointer to the beginning of the block. */
void *malloc(int num);
/* This function release a block of memory block specified by
address. */
void free(void *address);
```

Example of 1D dynamic array

```c
/* dynamic_1d_array.c */

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

int main(void) {

  int n;
  int* i_array; /* define the integer pointer */
  int j;

  /* find out how many integers are required */
  printf("Input the number of elements in the array:\n");
  scanf("%d",&n);

  /* allocate memory space for the array */
  i_array = (int*)malloc(n*sizeof(int));

  /* output the array */
  for (j=0;j<n;j++) {
    i_array[j]=j; /* use the pointer to walk along the array */
    printf("%d ",i_array[j]);
  }

  printf("\n");
  free((void*)i_array); /* free memory after use*/
  return 0;
}
```

Exercise

Calculate pi by Numerical Integration

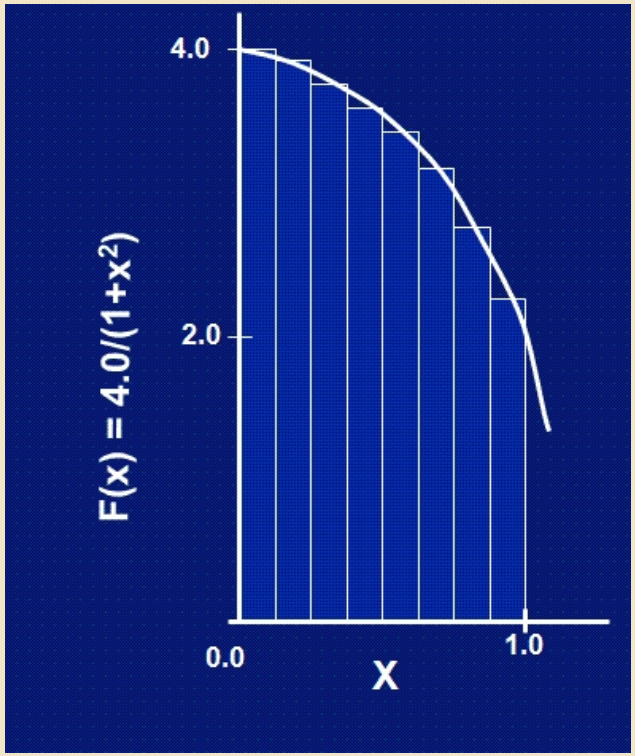
- We know that

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

- So numerically, we can approxiate pi as the sum of a number of rectangles

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Meadows et al, A "hands-on" introduction to OpenMP, SC09



Calculate pi by Numerical Integration

Algorithm 1 Pseudo Code for Calculating Pi

program CALCULATE_PI
  $step \leftarrow 1/n$
  $sum \leftarrow 0$
  do $i \leftarrow 0 \cdots n$
    $x \leftarrow (i + 0.5) * step; sum \leftarrow sum + 4/(1 + x^2)$
  end do
  $pi \leftarrow sum * step$
end program

- SAXPY is a common operation in computations with vector processors included as part of the BLAS routines

  $y \leftarrow \alpha x + y$

- Write a SAXPY code to multiply a vector with a scalar.

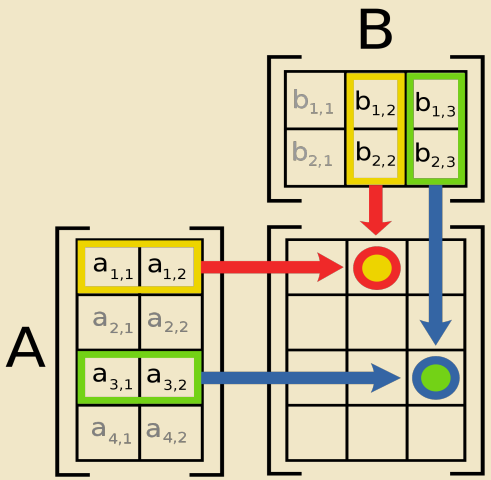
Algorithm 2 Pseudo Code for SAXPY

```
program SAXPY
    n ← some large number
    x(1 : n) ← some number say, 1
    y(1 : n) ← some other number say, 2
    a ← some other number ,say, 3
    do i ← 1 · · · n
        y_i ← y_i + a ∗ x_i
    end do
end program SAXPY
```

Matrix Multiplication

- Most Computational code involve matrix operations such as matrix multiplication.
- Consider a matrix **C** which is a product of two matrices **A** and **B**:

  Element *i,j* of **C** is the dot product of the i^{th} row of **A** and j^{th} column of **B**
- Write a MATMUL code to multiple two matrices.



Algorithm 3 Pseudo Code for MATMUL

program MATMUL
  $m, n \leftarrow$ some large number ≤ 1000
  Define a_{mn}, b_{nm}, c_{mm}
  $a_{ij} \leftarrow i + j; b_{ij} \leftarrow i - j; c_{ij} \leftarrow 0$
  do $i \leftarrow 1 \cdots m$
    do $j \leftarrow 1 \cdots m$
      $c_{i,j} \leftarrow \sum_{k=1}^{n} a_{i,k} * b_{k,j}$
    end do
  end do
end program MATMUL