History of C

- **C**
  - Evolved by Ritchie from two previous programming languages, BCPL and B
  - Used to develop UNIX
  - Used to write modern operating systems
  - Hardware independent (portable)
  - By late 1970's C had evolved to "Traditional C"

- **Standardization**
  - Many slight variations of C existed, and were incompatible
  - Committee formed to create a "unambiguous, machine-independent" definition → ANSI C
  - Standard created in 1989, updated in 1999

The C Standard Library

- C programs consist of pieces/modules called functions
  - Advantage: the programmer knows exactly how it works
  - Disadvantage: time consuming
  - Programmers will often use the C library functions
    - Use these as building blocks
    - Avoid re-inventing the wheel
  - If a pre-made function exists, generally best to use it rather than write your own
  - Library functions carefully written, efficient, and portable

Typical C Program Development Environment

- Phases of C++ Programs:
  - **Edit**
  - **Preprocess**
  - **Compile**
  - **Link**
  - **Load**
  - **Execute**

A Sample Program

```c
#include <stdio.h>

/* our first program in C */
int main()
{
    printf( "Welcome to C!\n" );
    return 0;
}

/* end program */
```

Comments

- Started by /* and ended by */
  - c++ compilers also recognize //
- Inserted to document programs and improve readability
- Do not cause any action to be performed
- Ignored by the C compiler

Program Analysis line 2

- #include <stdio.h>
  - Directive to the C preprocessor
  - Adds file containing data and/or functions
  - Processed before the program is compiled (hence preprocessing directive)
  - <stdio.h> is a library of functions for performing basic input / output operations.
Introduction to Programming

Program Analysis line 6

- printf( "Welcome to C!\n" );
- This is a statement
- It is a call to a function defined in <stdio.h> called printf

Printing Several Lines

- One printf statement can print several lines by using newline characters
  - e.g. printf("Welcome\nto\nC!\n");
- But that’s a stylistic horror! A better way:
  printf( "Welcome\n"  
  "to\n"  
  "C!\n" );

Important Point about Declarations

- You must make a declaration immediately following the left brace at the beginning of a function and before any executable statements or else you get a syntax error.

```c
main () {
    declaration section
    ...
    statement section
    ...
}
```

Most compilers will let you get away with declarations elsewhere in the function.

Printing Variables

- To print a variable, use the printf() statement:

```c
printf ("x:  %d\n", x);
```

- Format Specifier: Indicates the type of data to print. In this case, %d indicates decimal integer values.
- Name of variable to print. In this case, we print the variable x.

Example 1: Basic Arithmetic

```c
/* Illustrates Integer Variables */
#include <stdio.h>
main () {
    int x, y, z;
    x = 5;
    y = 10;
    z = x + y;
    printf ("x:  %d\n", x);
    printf ("y:  %d\n", y);
    printf ("z:  %d\n", z);
    return 0;
}
```

Note: Garbage values in C – the computer does not check to see if you initialize your variables

Rules for Identifiers

- Variable names are referred to as identifiers.
- Identifiers:
  - Can contain letters, digits, or underscores _
  - Cannot begin with a digit.
  - Can be any length but ANSI C only requires the first 31 characters are read. Therefore, you should never make your identifiers more than 31 characters. (Why?)
- Examples of valid identifiers:
  - int variable1, variable_2;
- Examples of invalid identifiers:
  - int [variable, variable1]
  - Remember, C is case sensitive, e.g. variable1 ≠ VARIABLE1
Basic Input
- Input: any user supplied data
  - Keyboard input, mouse input, etc.
  - scanf(): reads in keyboard input:
    - Located in the stdio.h library
    - Counterpart to printf().
  - Let’s take a look at an example…

Basic Input: scanf()
- If you want to read in data from the user, we use the scanf() function.
  - scanf() reads in data from the keyboard, and stores it in a specific variable.
  - Once stored in a variable, you can manipulate the data any way you want.
  - Very useful for creating interactive applications which require input from the user.

scanf() Syntax
- scanf has a very specific syntax, consisting of three parts:
  \[ \text{scanf}("%d", &\text{integer1}); \]
  - Format specifier: indicates the kind of data to expect. In this case, %d stands for decimal integer, e.g. whole numbers or integers.
  - Indicates where to store the data. In this case, if the user types 5, the number 5 is stored in integer1.
  - Note the & character. Without it, your computer may crash! (The & is the address operator in C.)

Adding Two Integers
- = (assignment operator)
  - Assigns a value to a variable
  - Is a binary operator (has two operands)
  \[ \text{sum} = \text{variable1} + \text{variable2}; \]
  - Variable receiving value on left
- printf( "Sum is %d\n", sum );
  - Similar to scanf
    - %d means decimal integer will be printed
    - sum specifies what integer will be printed
  - Calculations can be performed inside printf statements
    \[ \text{printf}( "Sum is %d\n", \text{integer1} + \text{integer2} ); \]

Scanf with Multiple Inputs
- You can use scanf to read in more than one variable at a time
  - For example,
    \[ \text{printf}( "Enter two numbers: " ); \text{scanf}("%d%d", &x, &y); \]
  - Reads two numbers from the user into x and y.
  - Note: When a computer program executes a scanf() statement, execution of the program waits until the user supplies the requested input.
Reserved Keywords

- **Keywords**
  - Special words reserved for C
  - Cannot be used as identifiers or variable names

<table>
<thead>
<tr>
<th>Keywords</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>auto</td>
<td>double</td>
</tr>
<tr>
<td>break</td>
<td>do</td>
</tr>
<tr>
<td>case</td>
<td>else</td>
</tr>
<tr>
<td>char</td>
<td>enum</td>
</tr>
<tr>
<td>const</td>
<td>float</td>
</tr>
<tr>
<td>continue</td>
<td>for</td>
</tr>
<tr>
<td>default</td>
<td>goto</td>
</tr>
<tr>
<td>do</td>
<td>if</td>
</tr>
<tr>
<td>double</td>
<td>if</td>
</tr>
<tr>
<td>enum</td>
<td>int</td>
</tr>
<tr>
<td>extern</td>
<td>long</td>
</tr>
<tr>
<td>float</td>
<td>register</td>
</tr>
<tr>
<td>for</td>
<td>return</td>
</tr>
<tr>
<td>goto</td>
<td>short</td>
</tr>
<tr>
<td>if</td>
<td>signed</td>
</tr>
<tr>
<td>include</td>
<td>sizeof</td>
</tr>
<tr>
<td>include</td>
<td>sizeof</td>
</tr>
<tr>
<td>include</td>
<td>struct</td>
</tr>
<tr>
<td>int</td>
<td>struct</td>
</tr>
<tr>
<td>int</td>
<td>switch</td>
</tr>
<tr>
<td>int</td>
<td>union</td>
</tr>
<tr>
<td>int</td>
<td>void</td>
</tr>
<tr>
<td>int</td>
<td>while</td>
</tr>
<tr>
<td>long</td>
<td>while</td>
</tr>
<tr>
<td>short</td>
<td>while</td>
</tr>
<tr>
<td>signed</td>
<td>while</td>
</tr>
<tr>
<td>sizeof</td>
<td>while</td>
</tr>
<tr>
<td>struct</td>
<td>while</td>
</tr>
<tr>
<td>switch</td>
<td>while</td>
</tr>
<tr>
<td>typedef</td>
<td>while</td>
</tr>
<tr>
<td>typedef</td>
<td>while</td>
</tr>
<tr>
<td>union</td>
<td>while</td>
</tr>
<tr>
<td>void</td>
<td>while</td>
</tr>
</tbody>
</table>

**Understanding Truth** (Revisited)

- In C (unlike life) truth is very simple.
  - 0 is False
  - Anything else is True
- For example:
  - -1 is true
  - 299 is true
  - 0 is false

**Truth Example**

```c
/* Understanding Truth */
#include <stdio.h>
int main()
{
    int x = -100;
    int y = 299;
    int z = 0;
    if (x)
        printf("x:  %d\n", x);
    if (y)
        printf("y:  %d\n", y);
    if (z)
        printf("z:  %d\n", z);
    return 0;
}
```

This is a completely valid program.

Output:

x:  -100
y:  299

Note: z is not printed because the if condition fails.

**Equality v. Assignment**

- Given:
  - if (grade = 100)
    - printf ("Perfect Score!");
- This statement does the following:
  - Grade is assigned the value 100.
  - Because 100 is true (i.e. non-zero!), the condition is always true.
- No matter what the student grade, it always says “Perfect Score!”

**#define**

- The #define preprocessor directive creates symbolic constants (it has another use we may talk about later)
  - `#define PI 3.14159`
- Would replace all subsequent occurrences of the symbolic constant PI with the numeric constant 3.14159, in the code

**#define (cont’d)**

- Used to avoid “magic numbers” in code
- Unlike variables, you should not change the value of a symbolic constant
- Should be used before your variable declarations
- Should be in ALL CAPS, except if you use Hungarian Notation
  - e.g. `#define I_MY_CONSTANT 3600`
Float Example

/* Float Example Program */
#include <stdio.h>
int main ()
{
float var1, var2, var3, sum;
var1 = 87.25;
var2 = 92.50;
var3 = 96.75;
sum = var1 + var2 + var3;
printf("Sum:  %.2f
", sum);
getchar();
return 0;
}

printf conversion specifications

- The format control string (the first argument to printf) contains:
  - conversion specifiers (%d, %e, %f)
  - field widths
  - precisions
  - other info we will not look at

Field width

- The exact size of the field in which data is to be printed is specified by a field width.
- If the field width is larger than the data being printed, the data will normally be right-justified within that field.
- Use a "-" sign before the number in field width to left-justify
- If the field width is smaller than the data being printed, the field width is increased to print the data.

Precision

- You can also specify the precision with which data is to be printed.
- Precision has different meaning -- for different types,
  - integer: minimum number of digits to be printed
  - float: number of digits to appear after the decimal point

Field width cont’d

- An integer representing the field width is inserted between the percent sign (%) and the conversion specifier
  - For example, %4d
    int YourAnswer = 1;
    printf("The answer %4d is correct!\n", YourAnswer);
    Will print the following
    The answer    1 is correct!

Precision cont’d

- To use precision, place a decimal point (.) followed by an integer representing the precision between the percent sign (%) and the conversion specifier (ie %.2f)
Character constant

- Characters are represented internally through int values.
- You can reference them by using single quotes around the character
  - E.g. ‘A’, ‘a’, ‘1’ or ‘m’
- What is actually happening is the computer stores these as numbers
  - The int value for ‘A’ is 65, and for ‘a’ it is 97, etc.

Declaring char variables

- Because they’re stored as ints, can use int gradeLetter;
- Better, clearer to use char gradeLetter;

Example using scanf

```c
#include <stdio.h>

int main () {
    char c;
    printf ("Enter a character:  ");
    scanf ("%c", &c);
    printf ("You entered:  %c\n", c);
    printf ("You entered:  %d\n", c);
    return 0;
}
```

Enter a character:  c
You entered:  c
You entered:  99

Note the difference in format specifiers.

Char functions: Reading and printing characters using getchar() and putchar().

getchar()

- Function contained in stdio.h
- Reminder: A function is like a small program other programs can use to perform actions (an example you’ve used is printf() )
- getchar() returns an int code for a character typed on the standard input (the keyboard)
  ```c
c = getchar();
```

How to read characters

- You can read char values with the scanf function
- C provides a function called getchar() which allows you to read characters one at a time from a line of input.
How to print a char

- Can use `printf` with a format control string of `%c`.
- For example,
  ```c
  printf( "The character is %c\n", c );
  ```
- Can use another function in `stdio.h` called `putchar`
- For example,
  ```c
  putchar( c );
  ```

**putchar()**

- Prints a character to the standard output (screen)
- Is also referenced in `stdio.h`

Using `getchar()` example

```c
#include <stdio.h>
int main()
{
  char c; /* declare character variable */
  /* read a character and store it in c */
  c = getchar(); /* print it twice, two different ways */
  printf("The character is %c\n", c);
  putchar(c);
  /* one character at a time so here's the newline */
  c = '
';
  putchar(c);
} /* end program */
```

**EOF character**

- C provides a special character which represents the end of a file. When processing a file, this character will be the last one encountered. (it is like a built in sentinel for file processing)
- The character is defined as `EOF`.
- You can also use the EOF char when interacting with a user. You must tell the user what to do.
- EOF is machine dependent.
  - In the windows environment it is `<control>-z`
  - Unix uses `<control>-d`

**EOF Example**

```c
#include <stdio.h>
int main()
{
  char c;
  printf("Enter characters terminated by <control>-z":
  c = getchar();
  while (c != EOF)
  {
    putchar(c);
    c = getchar();
  }
  return 0;
}
```

Using Pre-Packaged Functions

- Standard C includes lots of pre-packaged libraries:
  - `ctype.h`: character manipulation
  - `math.h`: mathematical functions
  - `stdio.h`: standard input/output
  - `stdlib.h`: random numbers, memory handling
  - `string.h`: string manipulation
  - `time.h`: date/time functions
Function Prototype

- Tells you what type of data the function is expecting, and what type of data the function returns.
  - Represents a communication protocol that enables two functions to "talk."
  - If you want to use a pre-built function, you need to learn to read prototypes.

An example: sqrt Function

- double sqrt (double);

  Return data type
  Function argument data type

- This function therefore accepts one double value, and returns one double value.

Math Library Functions

- Math library functions perform common mathematical calculations
  - #include <math.h>
- Format for calling functions
  - FunctionName (argument(s));
  - If multiple arguments, use comma-separated list
  - printf ("%.2f", sqrt (900.0));
  - Calls function sqrt, which returns the square root of its argument
  - Arguments may be constants, variables, or expressions

Using sqrt()

#include <stdio.h>
#include <math.h>

int main () {
    double value;
    value = sqrt(9.0);
    printf("%.f", value);
    return 0;
}

An Example: Creating your own Functions

#include <stdio.h>
int Square(int);

int main () {
    printf("%d ", Square(5));
    return 0;
}

int Square(int y)
    return y * y;

Function Template

- Use this template for adding functions to your program.

Function Prototype: must be declared at top of program.
return-value-type function-name (parameter-list);

Function Call: invokes the function.

Function Definition: contains the actual function code.

variable declarations;
statements of work;
return statement (optional):

Function Prototype: must be declared at top of program.
return-value-type function-name (parameter-list);

Function Definition: contains the actual function code.
Function Prototype v. Definition
● The Function prototype and the function definition must include the same data types. However,
  – Prototype: you do not need to include the names of parameters, but you must include the parameter data types.
    • e.g. int Square (int);
  – Definition: you must include the name and data type of all parameters.
    • e.g. int Square (int y)

Passing void
● If you use void in the parameter list, the function accepts no parameters
● Example:
  void printIntro (void);

Another Example
#include <stdio.h>
int Maximum (int, int, int);
main()
{
  printf("Maximum is: %d\n", Maximum(5, 7, 3));
}

Example: Argument Coercion
● Given the following function prototype:
  int Square(int);
● And, we invoke the function with a float value:
  printf("%d\n", Square(5.7));
● Because of argument coercion, 5.7 is truncated to 5.
● We therefore have a loss of data

Global v. Local Variables
● Global:
  – A variable declared outside any function
  – Can be referenced by any function in the program
● Local:
  – A variable declared inside a function
  – Can only be referenced within that function.
● If you have a local variable in a function and a global variable with the same name, the local one is used

Example
/* This program demonstrates global variables and scope */
#include <stdio.h>
void a (void);
void b (void);
int x = 1; /* Global Variable */
main () {
  printf("In main, x equals:  %d\n", x);
  a();
  b();
  printf("In main, x equals:  %d\n", x);
  return 0;
}
void a (void) {
  int x = 100;
  printf("In function (a), x equals:  %d\n", x);
}
void b (void) {
  printf("In function (b), x equals:  %d\n", x);
  x++;
}
pass by value

- In C when a parameter is passed to a function and the variable is modified by that function, the value does not change upon return to the calling function (there are exceptions to this including passing arrays and passing pointer types)
- This is not true in all programming languages.

Accessing elements with for loop

- Can use a for loop to print the contents of an array

```c
#define SIZE 5
int i = 0, myFirstArray[ SIZE ] ;
/* go through all element index */
for ( i = 0 ; i <= SIZE - 1 ; i++ )
{
    printf("%d", myFirstArray[ i ]) ;
} /* end for i */
```

Array Bounds

- Very Important: C does not provide array bounds checking.
- This means that your code can actually reference an element that does not exist!
- For example, suppose you have:
  ```c
  int students[10];
  students[100] = 45;
  ```
- 100 is very far outside your defined size (0..9)
- However, the compiler will not indicate an error. When you run the program, you will get very bizarre results. You may even crash your machine!

Call by Value v. Call by Reference

- Call by Value:
  - When you pass a variable, you pass a copy of the variable.
  - Changes to the copy variable do not affect the original variable.
  - In C, scalar variables, such as ints, floats, chars are passed via Call by Value.
- Call by Reference:
  - When you pass a variable, you pass a reference to the original variable.
  - Changes to the reference do affect the original variable.
  - In C, arrays are passed via call by reference.
Passing Arrays to Functions

- To pass an array to a function, we need a few new rules for function prototypes / definitions:
  - To specify an array parameter, indicate this with brackets. For example:
    ```c
    void MyFunction (float []);
    ```
    This function prototypes specifies that MyFunction () receives an array of float variables.
  - There is no need to include the size in the brackets. In fact, if you do specify the size, it is ignored.
  - As an alternative, it is recommended that you pass the array size as a separate variable. For example:
    ```c
    void MyFunction (float[], int);
    ```
    The first parameter indicates an array of floats; the second parameter indicates the size of the array.

Example

```c
#include <stdio.h>
#define SIZE 4
void MakeHot(int [], int);

int main () {
  int temp[SIZE] = {75, 65, 89, 72};
  int i;
  for (i=0; i<SIZE; i++) printf("%5d", temp[i]);
  printf("\n");
  MakeHot(temp, SIZE);
  for (i=0; i<SIZE; i++) printf("%5d", temp[i]);
  printf("\n");
  return 0;
}

void MakeHot(int thermo[], int size) {
  int i;
  printf("Making hot!\n");
  for (i=0; i<size; i++) thermo[i] += 10;
}
```

We pass the temp[] array to makeHot(). makeHot() then modifies the array. The function modifies the original array, and provides an example of call by reference.

Some Protection

- C provides a mechanism with which you can make an array non-modifiable
- Use the keyword “const” before the declaration in your function definition and prototype. For example:
  ```c
  Prototype:
  void ArrayPrint ( const int [], int);
  ```
  ```c
  Definition
  void ArrayPrint ( const int arrayToPrint[], int elementsCount )
  { ...
  }
  ```

Benefits of const over #define

- Like #define’s, const variables are not changeable
- However, const variables are known to the compiler!
- Can put them in watches, hold your cursor over them for values, etc.
- Great for debugging
Benefits of const over #define cont’d

- #defines are not known to the compiler at compile time because:
  - the preprocessor does simple text replacement of #define’s throughout your code before the compiler even gets a chance to look at it
  - That is the preprocessor does a “search and replace” on the name and replaces it with the value wherever it finds it
- Try looking at const variables and #define’s while debugging to see the difference.

Strings

- We have seen strings before:
  - When using printf, the stuff between the quotes is a string
    ```c
    printf("Hi, I am a string!\n");
    ```
  - When you break it down, a string like this is really an array of char’s
  - So how can we declare these strings?

Declaring strings

```c
char cMyFirstString[ ] = "Hello";
```

- This creates an array and initializes the elements of the array to the individual characters in the string
- Remember, since we don’t specify a size for the array, the initializing value determines the size
- Strings have a hidden character at the end that also gets added to the array…
- So the size of the array for “Hello” is six, not five, as the string termination character – known as a null character – is added.

Null character

- Strings have a special termination character at the end that signifies the end of the string
- The null character is a backslash followed by a zero. That is, the character looks like: ‘\0’
- All strings in C end with this character
- All arrays holding a string must be declared large enough to hold all the characters as well as the null character.

Declaring and initializing another way

- Remember how we initialized an integer array with initializers?
  ```c
  int iMyFirstArray[ 5 ] = { 0, 0, 0, 0, 0 };
  ```
- Can do something similar for strings, using initializers:
  ```c
  char cMyFirstString[ ] = { 'H', 'e', 'l', 'l', 'o', '\0' };
  ```
- This creates an array and initializes the elements of the array to the individual characters (note the null character at the end)

Declaring and Initializing (cont’d)

- As with integers, if you specify fewer initializing values than you have elements, all the rest are initialized to a value of 0. For examples,
  ```c
  char cMyFirstString[ 5 ] = { 'A' };
  ```
  would set the zeroth element to ‘A’ and the rest to 0!
- The string would be “A”, not “A0000”
  - i.e. the first 0 after the character is read as the null character

Declaring strings

```c
char cMyFirstString[] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

- This creates an array and initializes the elements of the array to the individual characters (note the null character at the end)
Accessing characters in an array

- Since a string is an array of characters, you can access the individual elements as you would with arrays of integers.
- For example, with

  ```c
  char cMyFirstString[10] = "Hello";

  cMyFirstString[0] is the character ‘H’ and
  cMyFirstString[2] is ‘l’
  ```

Printing Character Arrays

- You can print it the hard way...

  ```c
  for ( i = 0; sMyFirstString[i] != '\0'; i++ )
  {
    printf("%c", sMyFirstString[i] ) ;
  }
  ```

- ... or an easier way:

  ```c
  printf("%s\n", sMyFirstString );
  ```

- Note the new conversion specifier of %s for string!

Reading Input into Character Arrays

- You can read a string from the keyboard into a character array
- First, you must create an array large enough to hold your expected input

  ```c
  char sInput[20] ;
  ```

- Then use scanf with the %s specifier

  ```c
  scanf("%s", sInput );
  ```

- **NOTE THERE IS NO & BEFORE THE ARRAY NAME IN scanf!**
  - It is not needed for arrays!

Reading Input cont’d

- scanf will keep filling the array until you enter a whitespace character
- It is up to YOU to make sure that your array is large enough for the input
- If the array is too small for the input, scanf will continue to write beyond the length of the array
  - will most likely result in run-time errors
- Remember, you must leave a space for the null character!

String (char array) functions

- Strings are like char arrays in c.
  - They are terminated by the null char.
- You can use certain functions for manipulating strings by including the `string.h` header file
  - E.g. include <string.h>
- A useful function is strcpy
strcpy

- Allows you to copy one string, *including the terminating null character*, into another. E.g.:

```c
char destinationString[ 10 ] ;
strcpy( destinationString, "Class 1" );
```

- Copies “Class 1”, including space, into FIRST parameter, \( s\text{DestinationString} \).
- No bounds checking is performed
  - It is up to YOU to make sure the destination string is large enough (including the null character!)

random numbers

- Often we want our programs to generate random numbers.
  - games of chance
  - testing without user interaction
- c provides two functions which can be used to generate “pseudo-random numbers”

pseudo-random numbers

- When writing a program, you may not want totally random numbers. Sometimes you need your program to act operate deterministically.
  - Why?
- In other words you want the program to act the same each time you run the program.

rand()

- the stdlib.h library contains a function which will produce pseudo-random numbers
- Each time a call is made to the function, a value between 0 and RAND_MAX
  - RAND_MAX is determined by the compiler you are using.

rand() (continued)

- Each value is computed by taking the previous value returned by rand() and applying a formula to it.
- Because it is based on the previous number, any series will be identical if it starts with the same number.
- The series of numbers returned by rand() in a given environment will always be the same.

So what if we do want random numbers?
**srand (unsigned int)**

- srand() allows the programmer to seed the random number generator with any number.
  - srand(1) would start a deterministic list of numbers
  - srand(2) would start a different deterministic list of numbers
- Can we make random numbers yet?

**time.h**

- Another file in the c library is the time.h file. It contains various functions which are useful in dealing with time values.
- The function time() returns the current value of the system clock. Note: this value is not an integer but can be cast to an integer value.

**combining time() and srand()**

- By making the following call, we can simulate real random numbers
  
srand( (int) time (NULL) );

  Since the value of the internal clock is always changing the seed value will always change.

**using random numbers**

- Usually a programmer will write and debug the program with the same seed.
- Once the program is complete and tested, the rand function gets seeded with the current time.

**random number range**

- For this class, the easiest way to force your random numbers into a range is by using the % operator.
  - this is not the best method in real life
- To simulate the rolling of a die (i.e. get a number from 1 to 6) we could say:
  
  result = 1 + rand() % 6