C Programming

C Programming Language is non-object-oriented and fairly static but efficient.

It is a high-level language with variety of data types, functions, arrays, structures (records), decision and looping constructs and rich run-time library.

```
#include <stdio.h>

int main(int argc, char **argv) {
  int a, b;

  printf("Enter an integer: ");
  scanf("%d", &a);
  b = a * a;
  printf("%d is the square of %d\n", b, a);
}
```

argc: an integer count of command line arguments

****argv:** an array of pointers to the argument strings

main() returns an integer (zero for success, non-zero for failure-code)

→ Functions must be declared before being called

→ Declaration can go in a ‘header’ file (show return type, arguments and types with no body) = Prototypes in header file with semicolon at the end of each line

→ Fully defined function in a ‘.c’ source file contains code

Call-by-value

The value of variables are passed to funtions

```
#include <math.h>
void computeHypot(double a, double b, double hyp) {
  // This computes a result but the new value does not make it back
  // to the caller!!!
  hyp = sqrt((a * a) + (b * b));
}
double computeSquare(int num) {
  return num * num;
}
...

double result;
computeHypot(5, 6, result);
```

Call-by-reference

The address of (or reference to) a variable is passed to a called function (can be simulated using pointer variables)

```
#include <math.h>
void computeHypot(double a, double b, double *hyp) {
  // This computes a result but the new value does not make it back
  // to the caller!!!
 *hyp = sqrt((a * a) + (b * b));
}
...

double result;
computeHypot(5, 6, &result);
```

▼ gdb commands

(link1: https://mitny.github.io/articles/2016-08/gdb-command)

(link2: https://www.geeksforgeeks.org/gdb-command-in-linux-with-examples/)

gdb is the acronym for GNU Debugger. This tool helps to debug the programs written in C, C++, Ada, Fortran, etc. The console can be opened using the **gdb** command on terminal.

▼ x/<format> <addr>: examine value in <addr> as a <format>

Format letters:
- o – octal
- x – hex
- d – decimal
- u – unsigned decimal
- f – float
- i – instruction
- c – char
- s – string

Size format letters:
- b – byte
- h – halfword
- w – word
- g – giant (8 bytes)

▼ C Types

Basic Types

- int
- float, double
- char
- enum

Compound Types

- array
- struct

```
int main(int argc, char **argv) {
  int a, b, c;
  float myfloat = 5.7e-02; //initializer
  char option = 'n'; //initializer
  int odds[5] {1, 3, 5, 7, 9}; //initializer

  a = odds[0] + 2;
}
```

initializers: these set the variables to a value before the program or function starts executing

Enumerated type (enum)

enum: a group of named constants (Underlying representation is an integer and provides readability to applications)

Syntax:

```
enum <enumTag> = {<symbol1>,<symbol2>,…};
```

Example:

```
enum daysOfWeek = {Sunday, Monday, Tuesday,
        Wednesday, Thursday, Friday, Saturday};


if ((today >= Monday) && (today <= Friday)) {
  printf("Sorry, get to work!\n");
}
```

▼ Expressions

💡 Expressions can include: variables, constants, function calls, arithmetic operations, Boolean/ relational operations

💡 Expressions have a resulting value

💡 Expression values can be assigned to a variable or used immediately ‘inline’

Arithmetic Expressions

Large number of arithmetic operators

C Operators (partial)

Operators can be unary (with 1 operand) or binary (with 2 operands)

Operators have Precedence (Order of evaluation) and Associativity (How operator instances group). These properties can be over-riden with parenthesis.

++, -- (pre/post) increment/decrement

+, - (unary),

+, -, *, /, % Add, subtract, multiply, divide, mod

! – logical NOT

~ - bitwise NOT

==, !=, <=, <, >=, > - Relational tests opers

&, |, ^ Bitwise logical (and, or, not)

&&, || Logical Operators (and, or)

=, +=, -=, *=, /=, &=, |=, ^= Assignment

	precedence	associativity
1	() [] -- (post) ++ (post)	left-to-right
2	--(pre) ++(pre) unary - unary + ! ~	right-to-left
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	^	left-to-right
10	|	left-to-right
11	&&	left-to-right
12	||	left-to-right
13	assignments (=, +=, -=, *=, /=, etc)	right-to-left

```
#include <stdio.h>

int main(int argc, char **argv){
  int a = 10;
  int b = 5;
  int c = 2;

  d = b + c * 10; //25
  e = 2 * a / 10; //2
  f = c + b * a; //52
  e = e + f; //54
}
```

▼ C Statements

C Assignment Statements

Simple (=)

Compound (+=, -=, *=, etc)

C Selection Statements

if/then/else

```
Syntax:
  if (<relational test>)
    { statements }
  [else
    { statements }]

Example:
 if ((a > b) && (c < 10)) {
   printf("This is not good!\n");
 } else {
  printf("Ok, keep going\n");
 }
```

?:

```
Syntax:
  (<relational test>) ? expression : expression;

Example:
  printf((a>b) && (c<10)) ?
     "This is not good!\n" :
     "Ok, keep going\n" );
```

switch()

```
Syntax [Note, the '{}' around the case's code are optional]:
  switch (<var>) {
   case <val1>: {
        statements;
        [break;]
    }
   case <val2>: {
        statements;
        [break;]
    } …
   default:
  }
```

Iteration (loops)

while() / do…while()

for()

- Loops allow easy operation repetition (easier than assembler)

▼ Format

Format specifier: start with '%' and end with a letter to indicate type

Template: %[-,0]#.#<formatletter>

- Leading symbols like '-' or '0' indicate options like left justify or 0-fill
- First # is usually a field length
- Second # is usually a precision length

Specifier	Description
%[0]#d	Print integer in a field '#' bytes long [0-fill if lead 0 present]
%#.pf	Print float in a field '#' bytes long with 'p' digits of fraction
%[0]#x	Print hexidecimal value in a field '#' bytes long [0-fill if lead 0 present, use Uppercase A-F if specifier X is in Uppercase]
%c	Print a character
%[-]#s	Print a string in a field '#' bytes long. [Left justify if '-' present]

▼ C Pointers

Pointers are memory addresses. It tracks the type that a pointer variable points to.

Syntax:

```
<type> *<variableName>[ = &<declaredVariable>];
```

Example:

```
int *countAddress;
char *helloString = "Hello!";
```

Syntax:

```
<variable> = *<pointer>;
<pointer> = &<variable>;
```

Example:

```
count = 5;
countAddress = &count;
char aString[] = “This is a string.”;
char *stringPtr;
char firstChar;
// stringPtr points to same address as aString
stringPtr = aString;
firstChar = *stringPtr;
```

Pointers provide easy access to system structures (heap/memory allocation structures, process tables)

Pointers: Description / Use

- Pointers hold the address of a data item

→ Size of a pointer variable is the size of an architecture’s address (32-bit architectures, pointer variable 4 bytes & 64-bit architecture, pointer variable 8 bytes)

- Use

→ Passing large data between functions

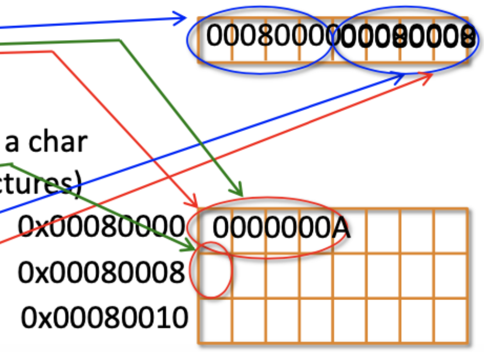
→ Returning multiple values from functions

→ Notational convenience (interchangeable with array variables)

Use ‘*’ in front of variable name – ‘*’ is descriptive indicating the variable name points to the named type

```
// pointerToMyInteger is a memory address of an int
// size of pointerToMyInteger  is 4 bytes (on 32 bit architecture)
// size of *pointerToMyInteger is 4 bytes (when compiler considers ‘int’s to be 32-bits)
int *pointerToMyInteger = 10;

// pMyChar is will hold the memory address of a char
// size of pMyChar is 4 bytes (on 32 bit architectures)
// size of *pMyChar is 1 byte
char *pMyChar;
pMyChar = (char *) malloc(sizeof(char));
```




```
struct _tempStruct {
   int amTemp;  // Temperature at 8 AM
   int pmTemp; // Temperature at 6 PM
   char location[20]; // Holds a city name
}

// pTempStruct is a memory address of an _tempStruct
// size of pTempStruct  is 4 bytes (on 32 bit architecture)
// size of *pTempStruct is 28 bytes (when compiler considers ‘int’s to be 32-bits).
// Also, cannot reference *pTempStruct as a whole. Must refer to fields within struct
struct _tempStruct *pTempStruct;
```

```
// Memory is reserved and the value 5 is placed there.
// The address returned from malloc() is placed in the variable pointerToMyInteger.
// The memory allocated by malloc is uninitialized.
int *pointerToMyInteger = malloc(sizeof(int));

// A variable is created called anotherMyInteger and the value 5 is stored there
int anotherMyInteger = 5;

// A pointer variable is created and initialized to point at the memory from anotherMyInteger
int *pointerToAnotherInt = &anotherMyInteger;
```

Pointer Use (Cont')

Setting a pointer variable involves generating the 'address' of a another variable

&: asks compiler to generate 'address' of the variable name

*: references the contents of the memory to which a variable points

```
int *pMyInt;
int oneValue, userValue = 10;   // Create two ints. Set userValue to 10
...
pMyInt = &userValue;  // pMyInt points to the storage where userValue was created
oneValue = *userValue; // Copies the memory contents of userValue (10) into one Value
```

```
// readline
//  fileDesc – File descriptor of an open file
//   buffer – (output) Holds pointer to a buffer allocated (by readline()) for returned file
data
//   length – (output) Number of characters read by function
int readline(int fileDesc, char **buffer, int *length) {…}

…
char *buffer;
int status, length;
int fd = open("File.txt", 'r');
…
status =readline(fd, &buffer, &length);
```

⇒ If pointers refer to an array element, they can be used to iterate through array (++, — operators will increment/decrement pointer by the size of the element to which it points)

```c
int myArray[] = {5, 10, 15, 20};
int *arrPtr = myArray;
int idx;

for (idx = 0; idx < sizeof(myArray)/sizeof(int); idx++)
{
  printf("Value: %d\n", *arrPtr);
  arrPtr++;
}
```

- C structures

C structures hold fields of differing types (used to collected related data)

Syntax:
```
struct <structTag> {
  <type> <name>;
  <type> <name>;
  …
}
```

Example:
```c
struct _employee {
  char firstName[40];
  char lastName[40];
  int employeeNumber;
  int manager;
  …
}
```

Syntax:
```
struct <structTag> <instanceName>;
```

Example:
```c
// Create an array of 1000 employee records
struct _employee persList[1000];
```

Syntax:
```
<instanceName>.<fieldName>
```

Example:
```c
nextEmp = 100;
strcpy(persList[nextEmp].firstName, 'Larry');
strcpy(persList[nextEmp].lastName, 'Boy');
persList[nextEmp].manager = 5;
```

▼ C macros

Macros are templates expanded by the C preprocessor.

Blind text substitution (no understanding of C syntax or semantics)

Can take arguments

Syntax:

```
#define <macroName> <subsText>
#define <macroName>(<parametersList>) <subsText>
```

Example:

```
#define  MAX_RECORDS  100
#define  SQUARE(A)  A*A
```

Syntax:

```
<macroName>
<macroName>(<argumentList>)
```

Example:

```
int a, b, c;
b = 5;
a = SQUARE(b);
c = SQUARE(b+1);    // Bad!
```

From last slide:

```
c = SQUARE(b+1);
    generates:
c = b + 1 * b + 1;    // which equals 2 * b + 1
```

Example:

```
#define SQUARE(A) ((A)*(A))
```

Now:

```
c = SQUARE(b+1);
     generates:
c = ((b+1) * (b+1));
```

▼ <stdio.h>

Functions:

- FILE *fopen(char *filename, char *mode);
- size_t fread(void *buffer, size_t size, size_t count, FILE *fptr);
- size_t fwrite(const void *buffer, size_t size, size_t count, FILE *fptr);
- int fprintf(FILE *fptr, const char *fmtString, ...);
- int fclose(FILE *fptr);
- int printf(const char *fmtString, ...);
- int scanf(const char *fmtString, ...);
- <lots more>

▼ <string.h>

Functions:

- char *strcpy(char *dest, char *src);
- char *strncpy(char *dest, char *src, size_t len);
- char *strcat(char *dest, char *src);
- char *strncat(char *dest, char *src, size_t len);
- void *memcpy(char *dest, char *src, size_t len);
- char *strchr(char *s, char c);
- char *strrchr(char *s, char c);
- <lots more>

▼ <math.h>

Functions:

- double sqrt(double x);
- double exp(double x);
- double pow(double x, double y);
- double sin(double x); // Also: cos, tan, asin, acos, atan

▼ Bit Operations

Low-level bit operations provide for manipulation of flags in data structures

Bit manipulation

C provides bitwise logical operators.

Useful for (Isolating bits or bit fields, setting bits, Unsetting (turning off) bits

C provides shift operators that help create masks

Bit Fields

Collection of contiguous bits

non-byte-aligned

field size not likely an integral number of bytes

& 논리곱
| 논리합
^ 배타적 논리합
~ 1의 보수
&= 비트 and 대입
|= 비트 or 대입
^= 비트 배타적 or 대입

▼ Function Inlining

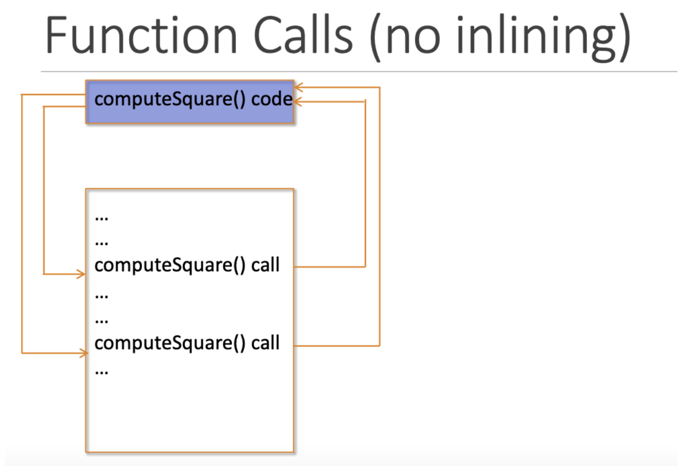
Functions add overhead to code:

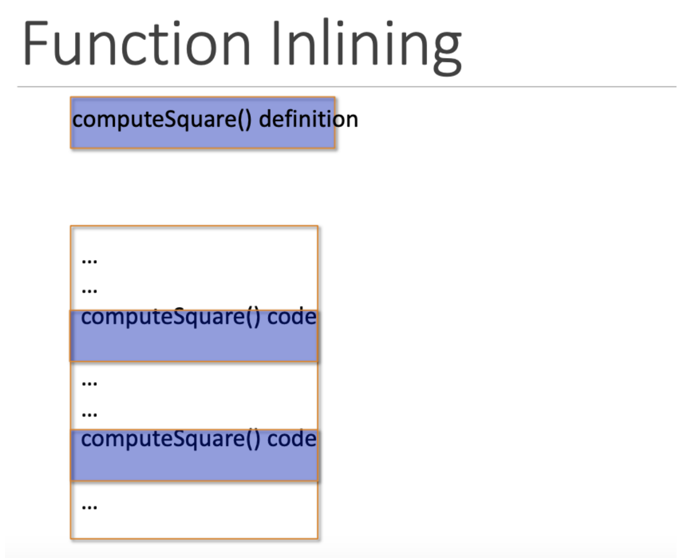
- Setting up arguments
- Transferring control to function
- Setting up return argument
- Transferring control back to caller

For efficiency, use inline with small functions:

- Inline causes a copy of the function code to be expanded at each call
- Costs memory for extra copies of code

- Saves run time during execution
- The inline keyword is a hint to the compiler





No inlining:

```
double computeSquare(double num) {
  return num * num;
}
...
int sqr = computeSquare(5);
```

Note 1:
Only difference is *inline* keyword ahead of declaration.

Note 2:
Function call does NOT change!

inlined:

```
inline double computeSquare(double num) {
  return num * num;
}
...
int sqr = computeSquare(5);
```

No inlining:

```
double computeSquare(double num) {
  return num * num;
}
...
int sqr = computeSquare(5);
```

Note 1:
To force the compiler to inline, specify the compiler directive listed here [only needed once/file.]

Forced inlined:

```
__attribute__((always_inline))
...
inline double computeSquare(double num) {
  return num * num;
}
...
int sqr = computeSquare(5);
```