Real-Time and Embedded Systems @ SIT

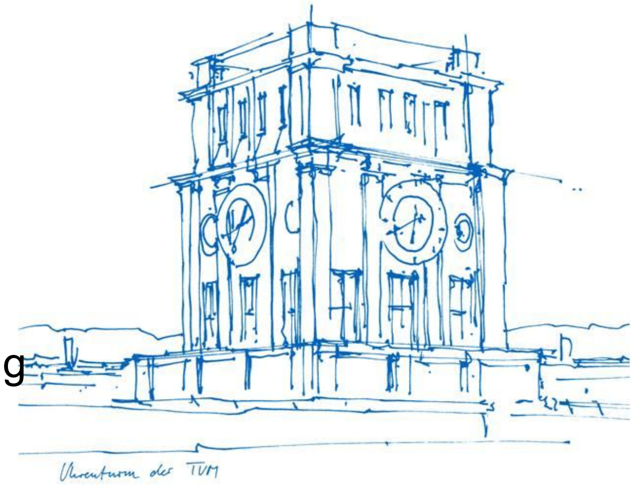
C Programming Language

Alexander Hoffman

Technical University of Munich

Department of Electrical and Computer Engineering

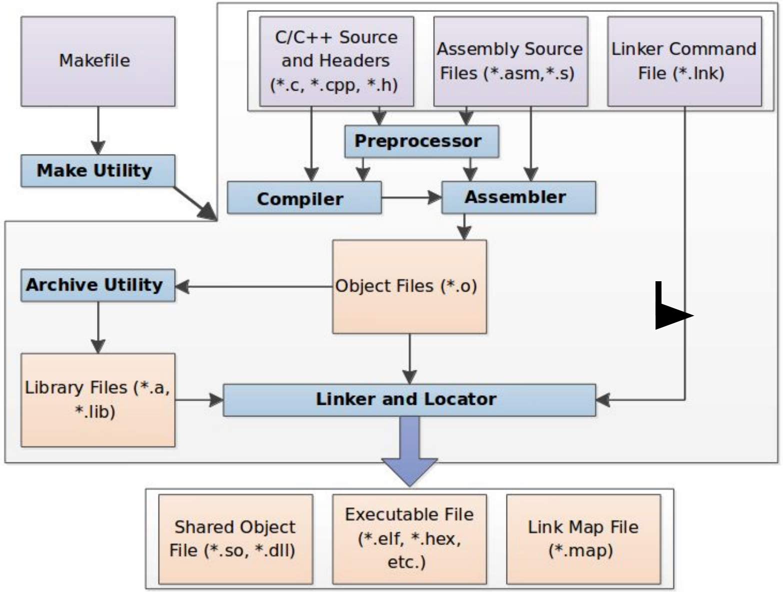
Chair of Real-Time Computer Systems



Outline

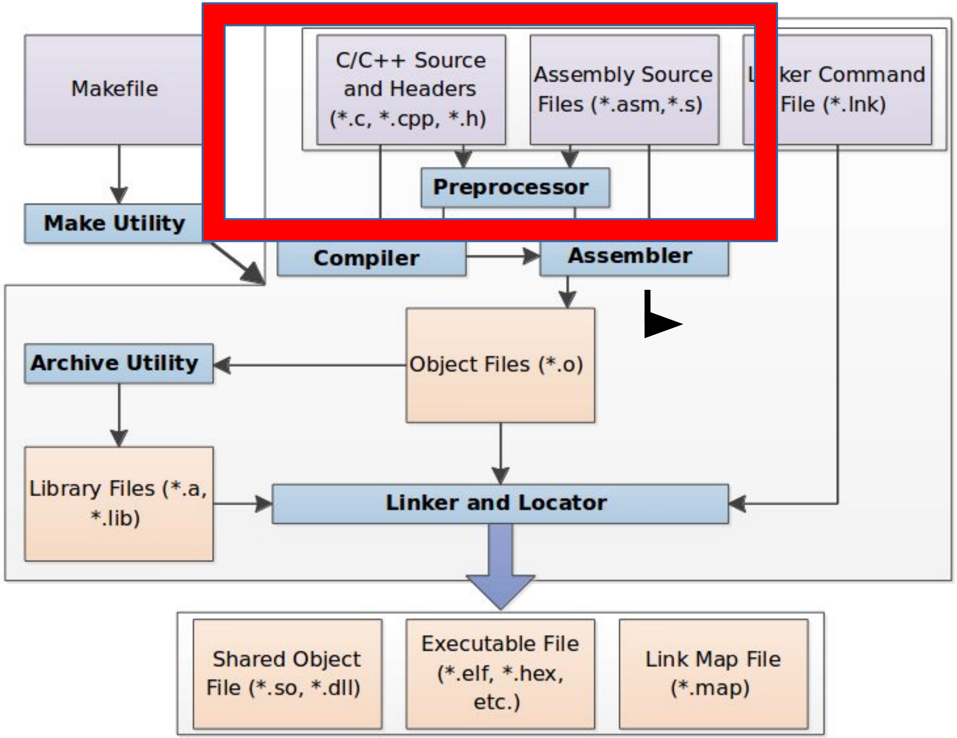
- Recap
- C – introduction
- Hello World Example
- Basic Syntax
- Memory Layout
- Data Types
- Variables
- Operators
- Conditional Operators
- Loops
- Functions
- Arrays
- Strings
- Pointers/Strings
- Memory Management
- Structures

Recap - Compilation

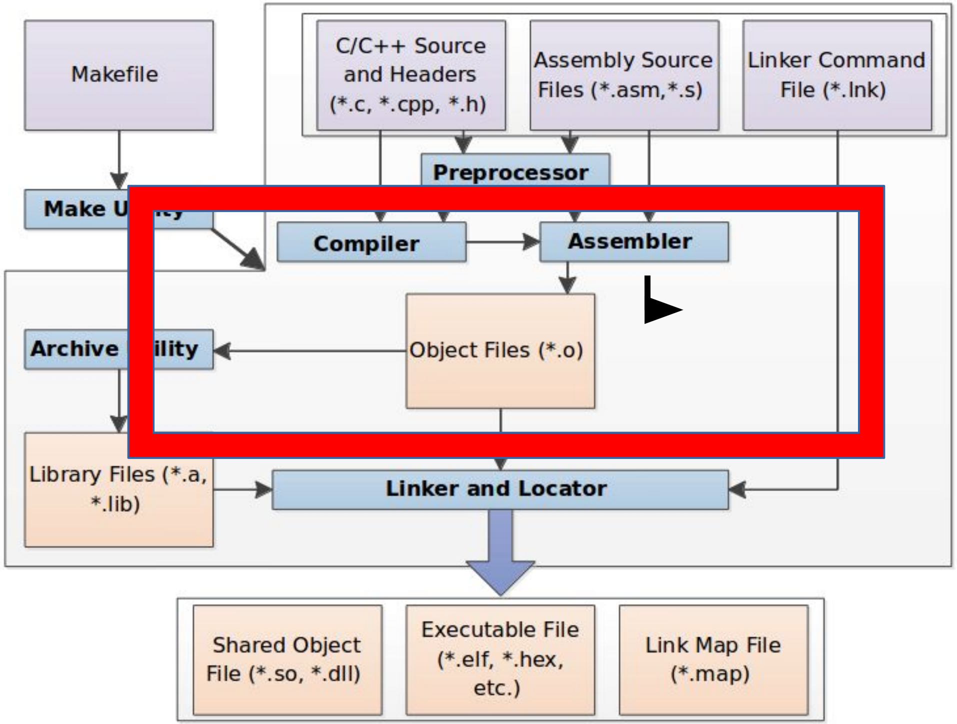


Check out a TUM tutorial that explains compilation in depth -> here

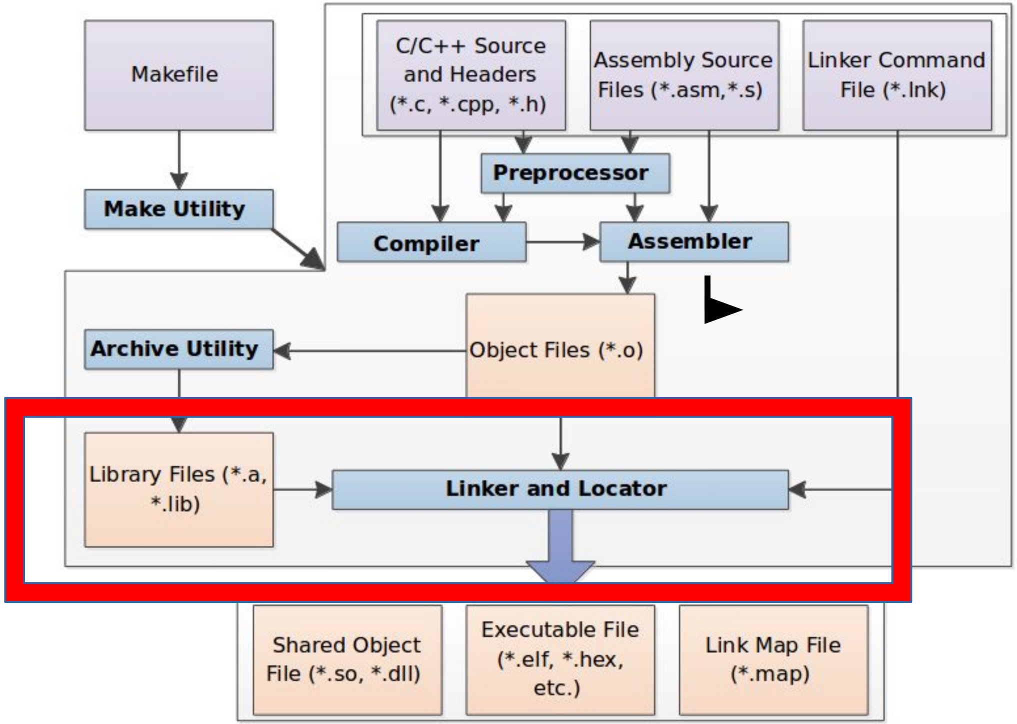
Recap – Compilation - Preprocessing



Recap – Compilation - Compiling



Recap – Compilation - Linking



C Programming Language

- General-purpose language (portable)
- Developed in 1972 by Bell Telephone Laboratories
- Highly structured language
- Very efficient (low overhead) and extremely fast (if used properly)
- Compiled language unlike interpreted languages (Python)
- Can handle low-level applications
  - Good for embedded systems (direct manipulation of hardware peripherals and memory)
  - OS kernels and device drivers (Linux/UNIX)
- My favourite language ;)

C Program Structure

- Is made up of:
  - Preprocessor directives
    - `#define`
    - `#include`
  - Functions
  - Variables
  - Statements & Expressions
  - Comments (never forget the comments!)

Hello World Example

```c
#include <stdio.h>

int main() {
    /* my first program in C */
    printf("Hello, World! \n");

    return 0;
}
```

Hello World Example

```c
#include <stdio.h>

int main() {
    /* my first program in C */
    printf("Hello, World! \n");

    return 0;
}
```

- Include directive (preprocessor)
  - Preprocessor effectively copy and pastes `stdio.h` into the line before compilation

Hello World Example

```c
#include <stdio.h>

int main() {
   /* my first program in C */
   printf("Hello, World! \n");

   return 0;
}
```

- Main function
  - Entry point into program (called by operating system)
  - Every program must have one

Hello World Example

```c
#include <stdio.h>

int main() {
    /* My first program in C.
       And this is a block comment */
    printf("Hello, World! \n"); // This is an in-line comment

    return 0;
}
```

- Comments
  - Document code and functions
  - Block comments (function explanations) or in-line comments (explaining a single line/logic)
  - Should not explain the how but the WHY
  - Helps other users use your code

Hello World Example

```c
#include <stdio.h>

int main() {
    /* My first program in C.
       And this is a block comment */
    printf("Hello, World! \n"); // This is an in-line comment

    return 0;
}
```

- Function call
    - Function is `printf`, found in `stdio.h`
    - Takes the input “`"Hello, World! \n"`”

Hello World Example

```c
#include <stdio.h>

int main() {
    /* My first program in C.
       And this is a block comment */
    printf("Hello, World! \n"); // This is an in-line comment

    return 0;
}
```

- Return value of `main`
    - Main returns an integer (int)
    - 0 represents success (usually)
        - !0 represents failure/error

C Syntax

- C interprets tokens, unlike languages like Python where spacing and whitespace are important
- Tokens can be:
    - Keywords - C programming language semantic
        - Eg. `return 0;` - return
    - Identifiers - name used to refer to variable, function etc.
    - Constants - values that can not be modified
    - Strings
        - Eg. `"Hello, World! \n"` - Both a constant and a string
    - Special symbols - C syntax to implement desired functionality, Eg. {}
    - Operators - trigger actions to variables and objects, Eg. +, -, ==
    - Comments*

C Tokens - Example

```
#include <stdio.h>

int main() {
   /* My first program in C.
      And this is a block comment */
   printf("Hello, World! \n"); // This is an in-line comment

   return 0;
}
```

Identifier, function name

Special symbol
Marks beginning of code block

Constant string

Identifier

Keyword
Defines data type

Special symbol

C Syntax - 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**

C Data Types

- Looking at keywords: int, char, short, double, float, void
- Determines how much memory should the variable will consume
  - Size can vary depending on compiler
  - Compiler implementation depends on: OS (if used) and hardware
  - Eg. `int` almost always the size of the platform, Ie. 64-bit x86 laptop has 64 bit `int`, 32 bit ARM microcontroller has 32-bit `int`
- Types classified into
  - Basic types - arithmetic types, Eg. `int` and `floats`
  - Enumerated types - `enum`
  - `void` - no value
  - Derived types - pointers, structures, functions etc.

C Data Types - Arithmetic Types

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	8 bytes	-9223372036854775808 to 9223372036854775807
unsigned long	8 bytes	0 to 18446744073709551615

`sizeof` operator returns size in bytes

C Data Types - Void Types

- Void functions
  - Do not return any value, ~return a void value which has no value
- Function arguments
  - Means the function takes no arguments

```
void my_function(void){
...
   // Do not need return statement
}
```

- `void` pointers - References an object but doesn't define a type. More on this later

C Variables

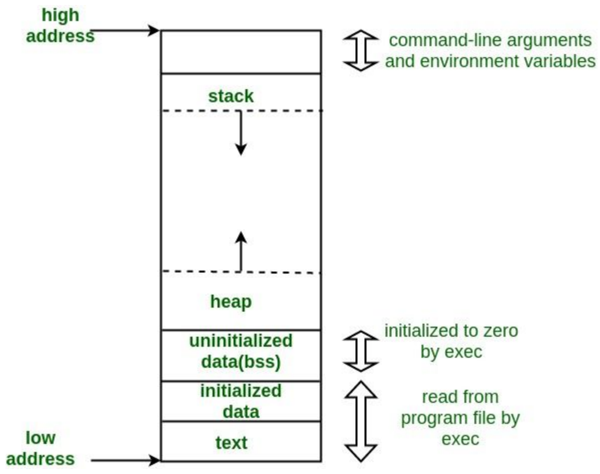
- Simply a human readable name given to a region in memory where some data is stored
- Variable type defines:
    - Size and layout in memory
    - Range of possible values
    - Operations that can be applied to variable

C Variables - Type Modifiers

- `const` - The value of the variable can't be changed after its definition
  - Prevents mistakes due to accidental modification
  - Allows for better optimization
  - Global const variables can be put into read-only memory
- `volatile` - Reads and writes to those variables have to happen exactly as often and exactly in the order as they happen in the program
  - Used for memory mapped I/O
  - Good for debugging, stops compiler optimizing out values

C Memory Layout

- To understand variables it is important to understand, even abstractly, how memory is structured



[geeksforgeeks.com]

Stack: contains temporary data for programs current stack frame

Heap: dynamically allocated variables, Ie. persistent across scopes in your program. As it is used the heap grows from a lower memory address towards the stack at a higher memory address.

Scope: currently executing block of code, eg. A function, for loop, etc.

C Memory Layout - Stack

Stack - This is the part of memory where local variables live, whose lifetime is bound to the current scope (e.g. function scope or inside a loop body).

- As scopes are stacked on top of each other, so are the lifetimes of the variables on the stack (the last one created will be the first one to be removed)
- Variable creation on the stack is very cheap (just increment the stack pointer)
- Stacks are often of limited size

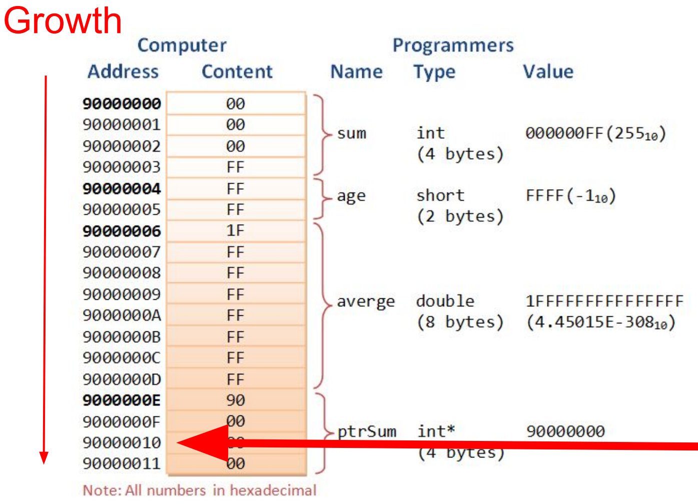
C Memory Layout - Heap

Stack - Memory can be allocated and deallocated at arbitrary times during the program execution (malloc/free)

- Allocator has to nd a free slot in memory -> (almost impossible to predict how long that will take) → rarely used in Real-Time Code
- Not bound to executing scope, Ie. variable can be used inside of multiple functions when proper methods are used

C Memory Layout Cont.

- Variables are stored sequentially (ideally) on the heap



[ntu.edu.sg]

Eg.

```
int sum = 255;
short age = -1;
```

Creates two variables sequentially in memory at addresses 90000000 and 90000004 respectively

Heap: dynamically allocated variables, Ie. persistent across scopes in your program. As it is used the heap grows from a lower memory address towards the stack at a higher memory address.

C Variables - Example on 64-bit x86 laptop

From before

- Data type tells us: size and layout, range of values, operators

```c
struct my_struct {
   int my_int;
   short my_short;
};

int sum = 255;
struct my_struct = {.my_int = 1, .my_short = 2};
```

\`sum\` is 32-bit value which can store between -2,147,483,648 to 2,147,483,647 and can use standard arithmetic operators such as \`++\`

C Variables - Example on 64-bit x86 laptop

From before

- Data type tells us: size and layout, range of values, operators

```c
struct my_struct {
   int my_int;
   short my_short;
};

int sum = 255;
struct my_struct = {.my_int = 1, .my_short = 2};
```

What’s this variable?!?!

\`sum\` is 32-bit value which can store between -2,147,483,648 to 2,147,483,647 and can use standard arithmetic operators such as \`++\`

C Structures

Allows for the definition of a variable type that holds several items of different or same types

- Definition:

```c
struct structure_type_name {

   //member definitions
   int int_member;
   ...
   short short_member;
} structure_instance_names;
```

- Creating and accessing:

```c
struct structure_type_name my_structure;
my_structure.int_member = 2;
```

  - Data type: `struct structure_type_name`
  - Variable name: `my_structure`

C Variables - Example on 64-bit x86 laptop

From before

- Data type tells us: size and layout, range of values, operators

```
struct my_struct {
    int my_int;
    short my_short;
};

int sum = 255;
struct my_struct = {.my_int = 1, .my_short = 2};
```

How would this look in memory?

`sum` is 64-bit value which can store between -2,147,483,648 to 2,147,483,647 and can use standard arithmetic operators such as `++`

C Variable - Example Cont.

```c
struct my_struct {
   int my_int;
   short my_short;
};

int sum = 255;
struct my_struct = {.my_int = 1, .my_short = 2};
```

- Data type `struct my_struct` tells us that the variable takes 6 bytes
  - `int`(4 bytes) + `short`(2 bytes)
- Data types of members tell us the ranges for each member
- `struct` keyword tell us what operators can be used
  - Eg. `my_struct++` is not possible but `my_struct.my_int` is

C Operators

Symbols that tell compiler to perform specific mathematical or logical operations

- Arithmetic operators: +, -, *, /, %, ++, --
- Relational operators: ==, !=, >, <, >=, <=
- Logical operators: &&, ||, !
- Bitwise operators: &, |, ^, ~, <<, >>
- Assignment operators: =, +=, -=, *=, etc.
- Misc operators: sizeof(), &, *, ?:

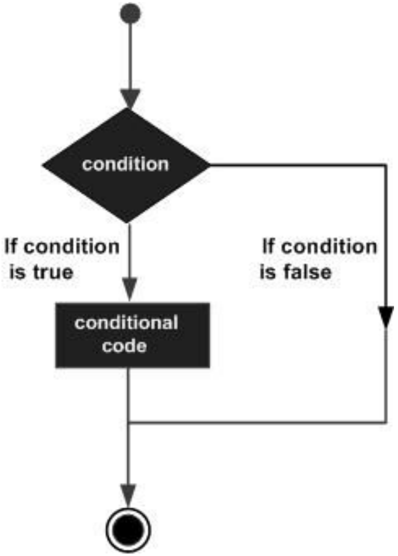
C Operators - Example

```c
int foo = 0b001; // Integer holds a value of 1
foo <<= 1; // Shift the value 1 bit to the right, ie. 0b010 == 2
foo++; // Increase value of foo by 1
int bar = (0b010 | 0b001); // Bitwise OR between 1 and 2

if (foo == bar){
   printf("foo equals bar\n")
}

>> foo equals bar
```

C Conditional Operators



[tutorialspoint.com]

Implements the conditional logic (decision making) of the program

C has 5 operators

- if - boolean expression
- if...else - if statement followed by an else statement
- nested if - if, else if,..., else if, optionally else
- switch - decide an action given a variable and a number of possible values for the variable
- nested switch - switch inside another switch

C Conditional Operators - Nested `if` Example

```
int foo = 2;

if (foo == 1){
   printf("One\n");
} else if (foo == 2){
   printf("Two\n");
} else {
   print("Something else\n");
}



>> Two
```

C Conditional Operators -`switch` Example

```c
int foo = 2;

switch (foo) {
case 1:
   printf("One\n");
   break;
case 2:
   printf("Two\n");
   break;
default:
   print("Something else\n");
   break;
}

>> Two
```

C Conditional Operators -`switch` Example

Switch cases “fall through” meaning that without a `break` statement the case below it will be executed.

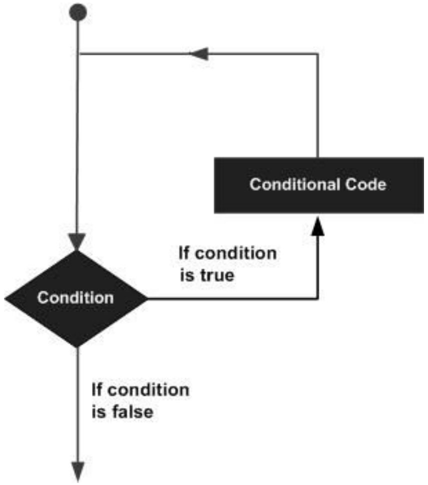
`default` is the equivalent to `else`.

```
int foo = 1;

switch (foo) {
case 1:
   printf("One\n");
case 2:
   printf("Two\n");
   break;
default:
   print("Something else\n");
   break;
}

>> One
>> Two
```

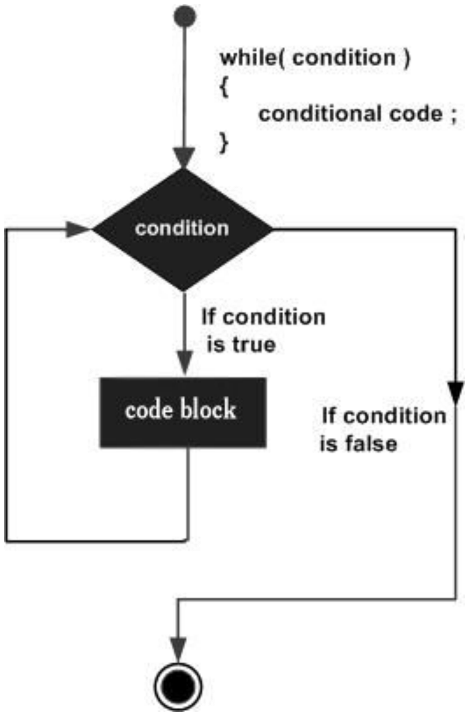
C Loops



[tutorialspoint.com]

- Used to execute a block of code several times
- Allows for more complex execution paths in a programm
- C offers three types of loops:
  - while - repeat while a condition is true, testing condition before executing loop body
  - for - execute loop body a specific number of times
  - do...while - while loop except the conditional statement happens after the loop body (always executes at least once)

C Loops - `while` Example



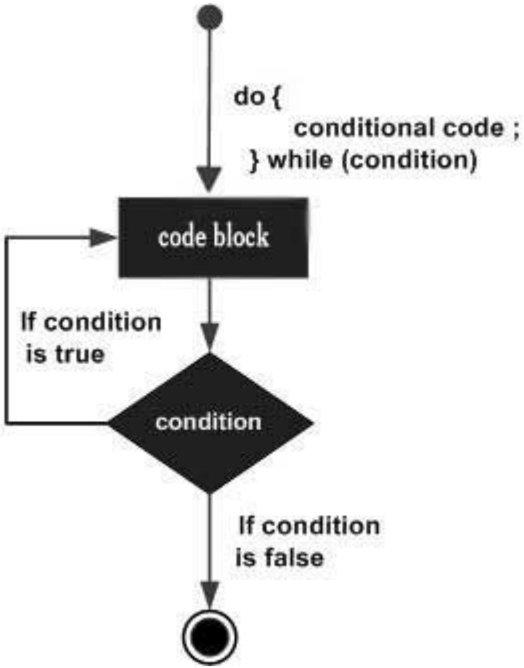
[tutorialspoint.com]

```c
int foo = 0;

while (1) {
    printf("%d\n", foo++);
}


>> 0
>> 1
>> 2
>> 3
```

C Loops - `do...while` Example



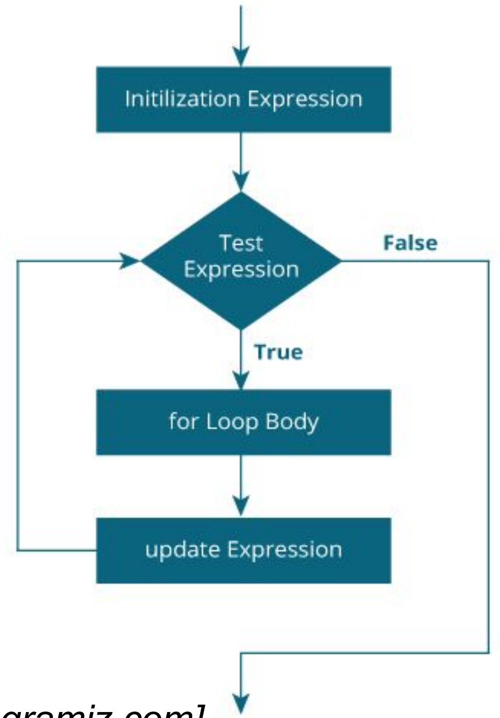
[tutorialspoint.com]

```c
int foo = 0;

do {
   printf("%d\n", foo++);
} while (foo > 1);

>> 0
```

C Loops - `for` Example



[programiz.com]

```c
int foo = 0;

for (int i = 0; i < 4; i++) {
   printf("%d\n", foo++);
}


>> 0
>> 1
>> 2
>> 3
```

C Loops - Control Statements

Used to change the execution sequence of a loop

- break - Breaks the execution of the loop or switch statement
  - Eg. terminate a for loop before the entire sequence has been iterated through
- continue - Terminates the execution of the remainder of the loop's body, continuing execution from the loop's condition
  - Eg. terminate the current execution of a for loop for a given iterator value, resuming at the next iterator value
- goto - Jumps execution to a labeled statement

C Control Statements - Example

```
int foo = 0;

for (int i = 0; i < 4; i++) {
    foo++
    if (i == 2)
        continue;
    printf("%d\n");
}

>> 1
>> 2
>> 4
```

```
int foo = 0;

while (1) {
    printf("%d\n", foo++);
    if (foo == 3)
        break;
}

>> 0
>> 1
>> 2
```

C Functions

- Group of statements (Eg. math ops, calls to other functions)
- Every program has at least one function, `main`
- Used to provide logical division and reusability to your code
  - Each function should perform one specific task
  - Careful division makes code more readable and easier to maintain
- Built in libraries provide many functions you can call, Eg. `printf`
- May have **multiple parameters** that are passed from the caller to the called function (the callee)
- Can return a value back to the caller when execution is completed

C Functions

C programs consist of **functions** that call each other

```c
void foo(void)
{
}
```

This minimal function called *foo* takes no parameters ((void)), does nothing ({}) and returns nothing (void).

`return_type function_name(parameter list){...}`

C Functions

C programs consist of **functions** that call each other

```
int foo(void)
{
    return 1;
}
```

foo now still takes no parameters but it now returns an *int* with a value of 1.

C Functions

C programs consist of **functions** that call each other

```
int foo(int bar)
{
    return bar * 2;
}
```

foo now a parameter *bar* of *int* type, still returning an *int*, only now it returns the parameter *bar*'s value x 2 instead of 1.

C Functions - Parameters

- Parameter variables are variables stored on the function's stack (~temporary memory for function execution)
- They can be manipulated in the function but unless returned using the `return` keyword (one value) they are lost once the function returns and the stack is discarded
- Passing pointers (discussed later) allows for you to modify variables in a function and have the changes persist after the function returns

Hello World Example - Revisited

```c
#include <stdio.h>

int main() {
   /* my first program in C */
   printf("Hello, World! \n");

   return 0;
}
```

- *main* is a function just like *foo*
- Every program needs to have exactly one main function
- Cannot be called by other functions in the program
- Entry point into program, called by operating system when starting program
- Return statement is optional, 0 (success) returned by default

C Functions - Declarations

- Compiler reads files top down
    - If a function is called it needs to of seen the function already to know it exists

```
#include <stdio.h>

int main(int argc, char* argv[])
{
	int foo = 3;
	int two_foo = get_double(foo);

	printf("Foo: %d\n", two_foo);
}

int get_double(int bar){
	return bar * 2;
}

>> implicit declaration of function ‘get_double’
```

C Functions - Declarations - Function Prototypes

- Compiler reads files top down
    - If a function is called it needs to of seen the function already to know it exists

```c
#include <stdio.h>

int main(int argc, char* argv[])
{
	int foo = 3;
	int two_foo = get_double(foo);
	printf("Foo: %d\n", two_foo);
}

int get_double(int bar){
	return bar * 2;
}
>> implicit declaration of function ‘get_double’
```

C Functions - Declarations - Function Prototypes

Place function prototype before function call

```
return_type function_name(parameter list);
```

```
#include <stdio.h>
int get_double(int bar);
int main(int argc, char* argv[])
{
      int foo = 3;
      int two_foo = get_double(foo);
      printf("Foo: %d\n", two_foo);
}

int get_double(int bar){
      return bar * 2;
}
>> Foo: 6
```

C Functions - Declarations - Function Ordering

Place function before function call

```
#include <stdio.h>

int get_double(int bar){
      return bar * 2;
}

int main(int argc, char* argv[])
{
      int foo = 3;
      int two_foo = get_double(foo);
      printf("Foo: %d\n", two_foo);
}
>> Foo: 6
```

C Functions - Declarations - Headers

- Header files include function prototypes for functions implemented in other .c files
- How you "include" library functions
- Eg.

```
#include <stdio.h>

int main(int argc, char* argv[])
{
   printf("Hello World\n", two_foo);
}
```

Preprocessor "copies and pastes" stdio.h into the #include line

C Functions - Declarations - Headers

Looking into `stdio.h` we see the following

```c
   This function is a possible cancellation point and therefore not
   marked with __THROW.  */
extern int printf (const char *__restrict __format, ...);
/* Write formatted output to S.  */
extern int sprintf (char *__restrict __s,
            const char *__restrict __format, ...) __THROWNL;

```

Along with a LOT of other function prototypes that are prototypes of functions implemented in `stdio.c`

As include is at top of file our compiler knows that `printf` exists

C Headers

- Don’t have to repeat function prototype over and over and over
- Either:
    - `#include “my_header.h”` - compiler looks in project source files
    - `#include <stdio.h>` - compiler looks into system libraries
- Include file everywhere the functions that it defines are needed, simple!
- Standard libraries, Eg. `stdio.h`, are usually shipped with a compiler and thus have a target hardware platform
    - `stdio.h` on your laptop and on an embedded system could have very different functionalities!

C "Advanced" Data Types

Before going into the topics that are the most confusing just remember:

- Everything in C is simply some human-readable identifier that is used to reference some location in memory after compilation
- What is at that location in memory is told to the compiler using data types
  - Compiler can do some loose sanity type checking
- C allows the user to access wherever he/she wants in memory
- C does not keep track of data types, the programmer should know what data type is stored where in memory

C Arrays

- Data structure that tells the compiler “Here is a fixed number of X data type variables stored sequentially in memory”
- Key is that an array is a continuous memory region
- Each item is accessed via an index

Declare:

```
type array_name[array_size];
```

Access:

```
array_name[index];
```

C Arrays - Example

- Can be initialized with values or zeroed out
  - WARNING: if not initialized then values are just random junk from memory!
- Accessed using [ ] notation

```c
int main(int argc, char* argv[])
{
   int my_int_array[20] = {0}; // Set all items to 0
   my_int_array[0] = 1; // Set first element to 1
   my_int_array[19] = my_int_array[0]; // Set last element to equal first

   double my_doubles[5] = { 123.45, 22.22, 1.0, 99.9, 2.0 };
}
```

C Arrays - Example

[ ] notation is actually just saying:
“ From the memory location at the beginning of the array, go index * sizeof(data type) memory locations further and return the value you find there”, although this is dependent on the compiler and advanced concepts such as memory alignment. Ignore this for now.

```
int main(int argc, char* argv[])
{
   int my_int_array[20] = {0}; // Set all items to 0
   my_int_array[5] = 1; // Set first element to 1
   // Same as
   *(my_int_array + 5) = 1; // * means were are looking at contents of a memory address, more on
this soon
}
```

C Arrays - Strings

- C doesn't have native string types like other languages, Eg. Python's `str`
- A "string" in C is simply an array of `char` (1 byte ASCII values) with a termination character `` `\0` `` at the end
- Functions that operate on strings just iterate through memory one `char`/byte at a time until they encounter the termination character
- Use array notation to be declared:
    - `char my_string[12] = “Hello world”; // Note length accounts for ‘\0’ char`
    - `char my_string[] = “Hello world”; // Compiler computes require array length`

C Arrays - Strings - Example

```c
#include <stdio.h>

int main(int argc, char* argv[])
{
    char hello_string[5] = { 'h', 'e', 'l', 'l', 'o'};
    char world_string[] = " world";
    printf("%s %s\n", hello_string, world_string);
}

>> hello world
```

Now for the most powerful feature of C:

POINTERS

(also that which confuses students the most)

C Pointers

- Remember: variable is simply a data type at some memory location
- A pointer is a variable that stores a memory location
  - Denoted using *
  - Eg. “`int *foo`” is a pointer (stores memory location) and at the location it points to is an `int` data type
  - Pointer data type just tells compiler what to expect (size of) at that memory location
- & ampersand operator can be used to get memory location of a variable
  - Eg. “`int *foo = &bar`” stores the memory location of the variable `bar` in the pointer `foo`. `bar` should be an `int`.

C Pointers

- You can access the contents of a pointer using the * operator
    - The operator does the following: “It goes to the memory location and reads/writes the number of bytes given by the data type”
    - Eg. `*foo = 2`

      Would go to the memory location that `foo` has stored (the `bar` variable’s address) and stores the `int` 2 (4 bytes) at that location.

C Pointers - Example

```
#include <stdio.h>

int main(int argc, char* argv[])
{
   int foo = 2; // foo is set to a value of 2
   printf("%d\n", foo);

   int *bar // bar is a pointer that points to an integer value
   bar = &foo; // store the address of the foo variable in bar
   printf("%p\n", &foo);
   printf("%p\n", bar); // Same

   *bar = 4 // The int value at the location bar has stored (foo) is set to 4
   printf("%d\n", foo);
}
>> 2
>> 0x7ffc0c08d37c
>> 0x7ffc0c08d37c
>> 4
```

C Pointers

- Allow you to pass variables to functions and have the values remain after the function returns
  - Changed directly in memory (heap) and not on the function's stack
- Only way to allocate memory on the heap using `*alloc` functions

```
int * p = malloc(10* sizeof(int));
*p = 5;
free(5);
```

C Memory Management

Pointers are how you store references to memory allocated on the heap.
But what is this allocation?

- Memory on heap must be allocated before being used
- Memory SHOULD be free’d when no longer needed
- Memory allocation is done using the `*alloc` functions from `stdlib.h`
- User needs to specify the number of bytes to be stored
- Accessing allocated memory is done using pointer, pointer data type helps compiler know what is to be accessed at memory location

C Memory Management - Alloc Functions

- void *calloc(int num, int size) - allocates an array of **num** elements, each of **size** bytes
- void *malloc(int num) - allocates **num** bytes
- void *realloc(void *address, int newsize) - re-allocates memory at **address**, extending it up to **newsize.** Might move memory region.
- void free(void *address) - releases the block of memory at **address**

- All functions return **void *** meaning they have no type information
- User should type case (give return pointer a type) returned memory pointer. Eg. `int *foo = (int *)malloc(sizeof(int));`
- Allocation returns NULL is failed.

C Structures - In depth

- A user defined collection of multiple data items
- Allows for well structured data objects
  - Thus enabling programming paradigms such as object orientated programming
- Defined using the `struct` keyword
- Accessed using the member access operator `.`
  - Eg. `my_struct.my_int_member = 2;`
- Good way to pass in and return lots of data to functions
  - `return` can only return one variable (put lots in a `struct` ;))

C Structures

- Can be allocated dynamically just like other variables
    - Eg.

```c
struct my_struct {
   int int_member;
   short short_member;
};
struct my_struct *my_struct_pointer =
    (struct my_struct *)malloc(sizeof(struct my_struct));
```

Pointer to my structure.
Type is “`struct my_struct *`”, this tells the compiler that a “`struct my_struct`” is at the memory location stored in “`my_struct_pointer`”

Structure definition:
What it’s called and what’s
In it!
Referenced using “`struct my_struct`”, don’t forget `struct` keyword

C Structures

- Can be allocated dynamically just like other variables
  - Eg.

```
struct my_struct {
   int int_member;
   short short_member;
};
struct my_struct *my_struct_pointer =
    (struct my_struct *)malloc(sizeof(struct my_struct));
```

Type casing return value from `malloc`.
So we know what is at memory location

`malloc` needs the number of bytes to allocate, sizeof returns the number of bytes that "`struct my_struct`" requires (4 (`int`) + 2 (`short`)) = 6

C Structure Pointers - Accessing Members

- Tricky!
- Pointer points to structure in memory
- Getting member values requires two steps:
  - Dereferencing pointer to give structure variable
  - Need to use member access operator (**.**) to get member values from dereferences structure
- Copying my_int_member to another variable equates to:

```
int foo = (*my_struct_pointer).int_member;
```

C Structure Pointers - Accessing Members

```
int foo = (*my_struct_pointer).int_member;
```

Dereference pointer FIRST (brackets)

Access member of now dereferenced pointer

Setting variable is similar:

```
(*my_struct_pointer).int_member = 5;
```

C Structure Pointers - Accessing Members

This method is messy and can get confusing

Luckily C has an easier way!

The (**->**) operator does these two steps in one. It dereferences the pointer and accesses a member of the dereferenced structure.

Eg.

```
int foo = my_struct_pointer->int_member;
my_struct_pointer->int_member = 5;
```

C Structure Pointers - Accessing Members - Example

```c
struct my_struct {
   int int_member;
   short short_member;
};
struct my_struct *my_struct_pointer =
    (struct my_struct *)malloc(sizeof(struct my_struct));

if (my_structure_pointer == NULL)
   return -1; // Allocation failed! Return error code

my_structure_pointer->short_member = 5;
my_structure_pointer->int_member = (int)my_structure_pointer->short_member;
```