Introduction to C

```
module = Module(
    code="ELEE1147",
    name="Programming for Engineers",
    credits=15,
    module_leader="Seb Blair BEng(H) PGCAP MIET MIHEEM FHEA"
)
```

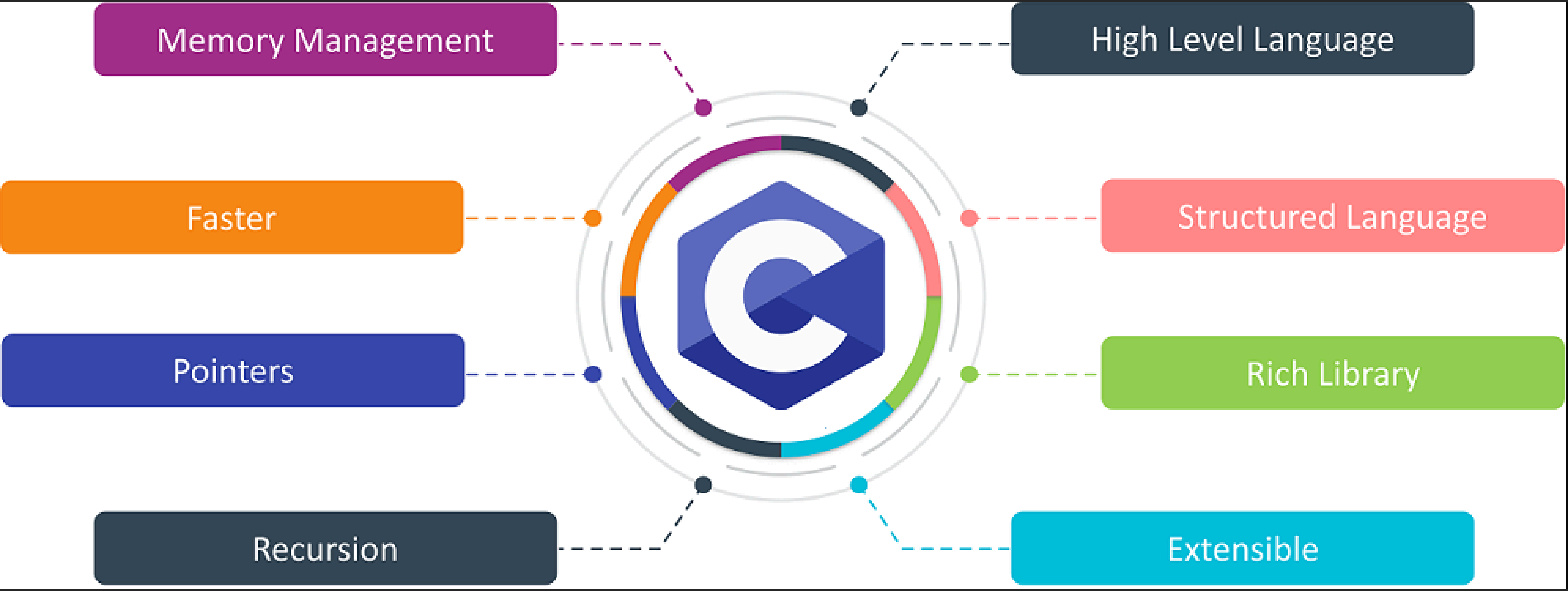
Download as a PDF

Why C?

Developed by Denis Ritchie [1941-2011] in 1972

- Low-level access to memory
- A simple set of keywords
- A clean style
- Suitable for system programming:
  - operating systems
  - compiler development
- Procedural and structured programming
- Portable across various platforms
- Combines low-level hardware control and high-level language convenience

```c
#include <stdio.h>

int main(){

  printf("Hello World!\n");

  printf("Goodbye World!");

  return 0;
}
```



Memory Management

High Level Language

Faster

Structured Language

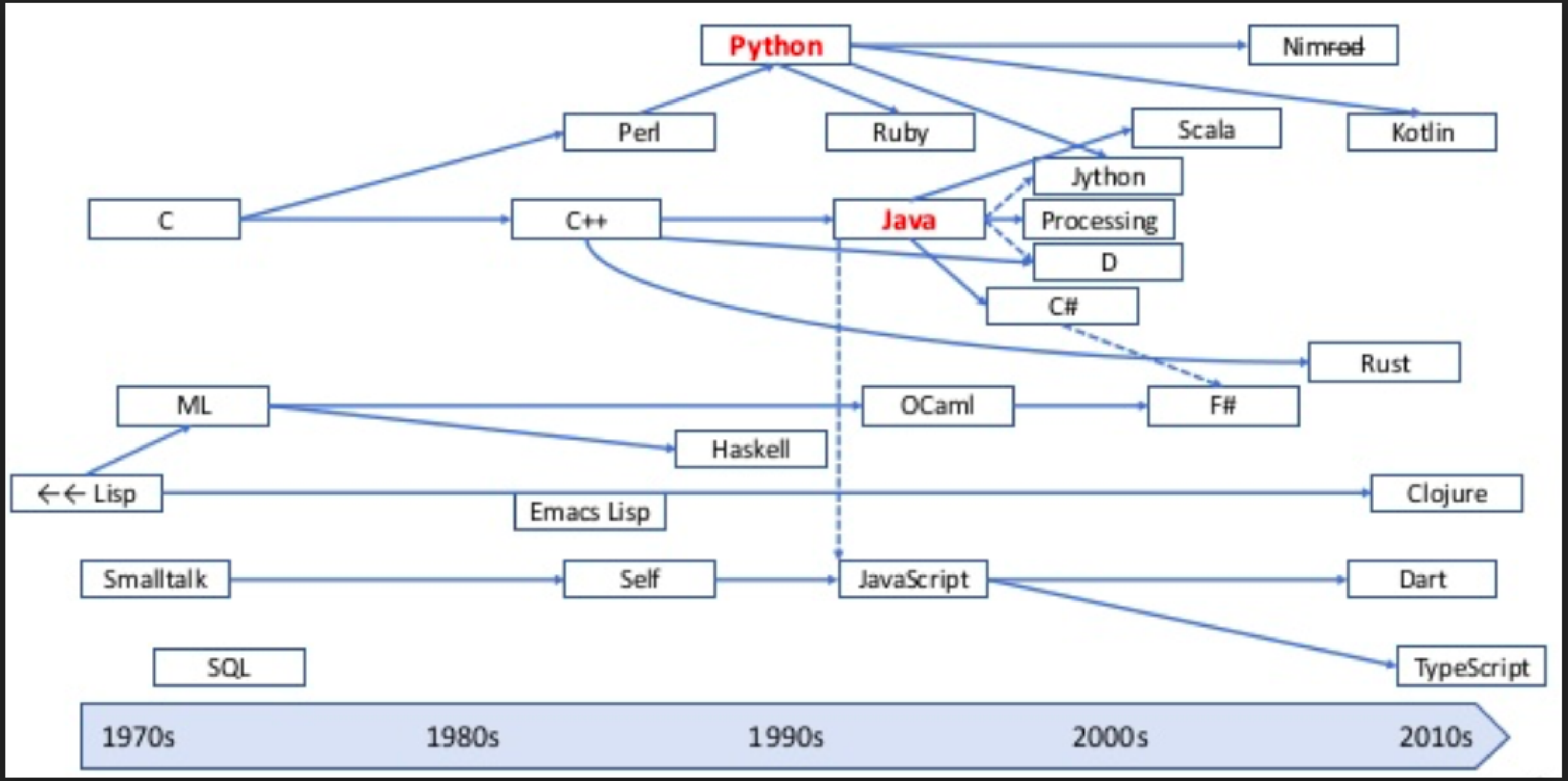
Pointers

Rich Library

Recursion

Extensible

C God's programming language*



*Shreiner D. 2010. OpenGL programming guide : the official guide to learning OpenGL, versions 3.0 and 3.1. 7th ed. Upper Saddle River, Nj: Addison-Wesley.

First Program in C

```
$ mkdir Learning_C && cd Learning_C
$ mkdir Helloworld && cd Helloworld
$ touch helloworld.c
$ <nano/vim/vi> helloworld.c
```

- Single line comments

```c
// library or header file that contains  standard input/output operatioins

#include <stdio.h>

/*
  main() function every C program must have a main,
  it has a returnable 'int' this is for exit codes
*/
int main(void) // void means no input argument
{
    printf("Hello World!\n");
    printf("Goodbye World!\n");
    return 0; // return exit code 0, no error
}
```

- directive, tells the preprocessor to include the contents of a specified file.

```c
// library or header file that contains  standard input/output operatioins

#include <stdio.h>

/*
  main() function every C program must have a main,
  it has a returnable 'int' this is for exit codes
*/
int main(void) // void means no input argument
{
    printf("Hello World!\n");
    printf("Goodbye World!\n");
    return 0; // return exit code 0, no error
}
```

- Multi-line comments

```c
// library or header file that contains  standard input/output operatioins

#include <stdio.h>

/*
  main() function every C program must have a main,
  it has a returnable 'int' this is for exit codes
*/
int main(void) // void means no input argument
{
    printf("Hello World!\n");
    printf("Goodbye World!\n");
    return 0; // return exit code 0, no error
}
```

- All **C** programs need a `main()` function as it's entry point

```c
// library or header file that contains  standard input/output operatioins

#include <stdio.h>

/*
  main() function every C program must have a main,
  it has a returnable 'int' this is for exit codes
*/
int main(void) // void means no input argument
{
    printf("Hello World!\n");
    printf("Goodbye World!\n");
    return 0; // return exit code 0, no error
}
```

- Body of the code that is executed, wrapped in braces `{}`

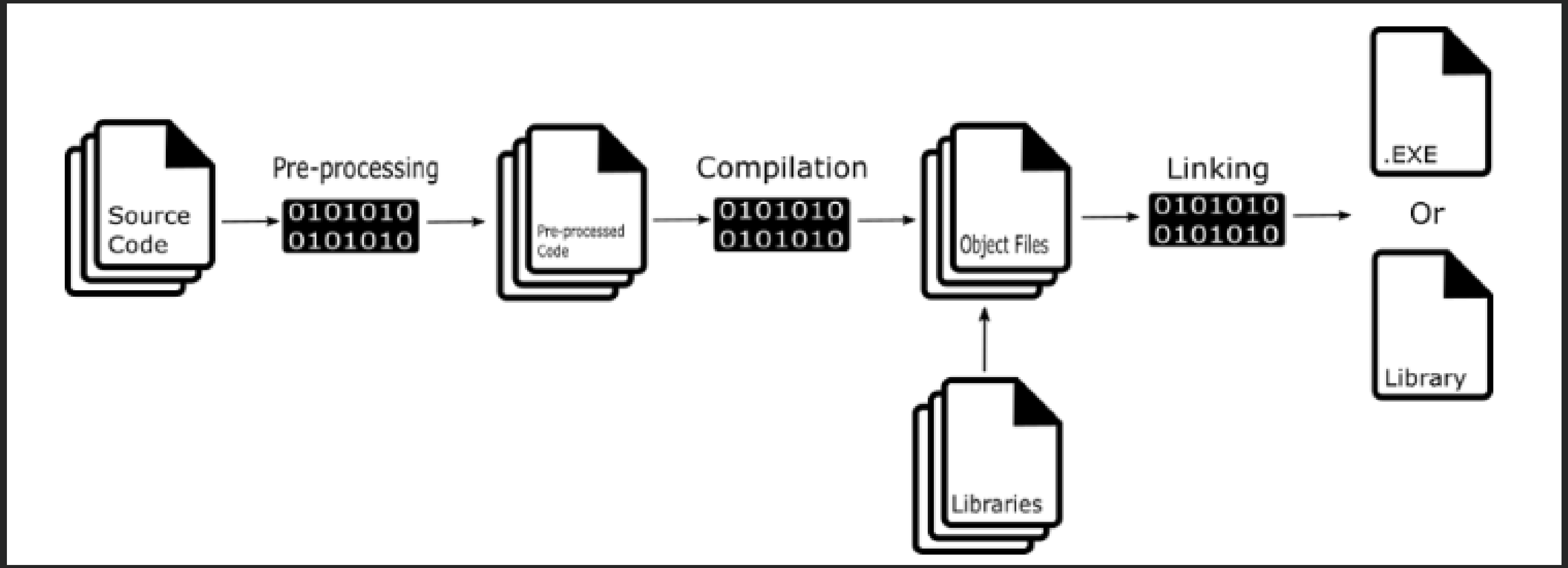
```c
// library or header file that contains  standard input/output operatioins

#include <stdio.h>

/*
  main() function every C program must have a main,
  it has a returnable 'int' this is for exit codes
*/
int main(void) // void means no input argument
{
    printf("Hello World!\n");
    printf("Goodbye World!\n");
    return 0; // return exit code 0, no error
}
```

Compile the code

We are going to use `gcc` compiler to compile our `c` code;

```
$ gcc helloworld.c -o helloworld.exe
```

- first argument is the source file[s], `helloworld.c`
- `-o` means output file, `helloworld.exe`
- The file extension in linux can be left blank or called whatever you want.



Execute the code

- As we are using a terminal, we must prepend the newley created file with `./`.
  - The `.` denotes the current directory.
- Since we want to run a file in our current directory which is not our `$PATH`
  - You need the `./` bit to tell the **shell** where the executable is.

Output:

```
$ ./helloworld.exe
Hello World!
Goodbye World!
```

Header Files

These files contain all scaffolding code that your `main()` will use as we do not want to overpopulate with excessive lines of code for readability.

Computers used to be too slow to compile a whole program in one single mega-blob where all the functions and types were visible.

To solve this, programs are split into c/h files which are each individually compiled into a machine code file (the ‘object’ file), and then linked together into an `exe` or `dll`.

- Create a new header file:

```
$ touch usefulfunctions.h
$ <nano/vim/vi> usefulfunctions.h
```

- `usefulfunctions.h`

```
#ifndef USEFULFUNCTIONS_H_   /* Include guard */
#define USEFULFUNCTIONS_H_

int sqr(int x);  /* An example function declaration */

#endif // USEFULFUNCTIONS_H_
```

- Create a new source file:

```
$ touch usefulfunctions.c
$ <nano/vim/vi> usefulfunctions.c
```

- `usefulfunctions.c`

```
#include "usefulfunctions.h"  /* Include the header (not strictly necessary here) */

int sqr(int x)     /* Function definition */
{
    return x * x;
}
```

Revist helloworld.c

We are modifiying the code to use our custom library:

```c
#include <stdio.h> /* searches system header file directories */
#include "usefulfunctions.h" /* notice "" searches current directory */

int main(void) /* void means no input argument */
{
    printf("Hello World\n!");  /* using standard ouput function to printf()*/
    printf("%d\n",sqr(255));
    printf("Goodbye World!\n");

    return 0; /* return exit code 0, no error */
}
```

Compile and run

We need to source all files needed to build our modified program.

Remember the header file points to the function in the `usefulfunctions.c` file.

```
$ gcc helloworld.c usefulfunctions.c -o helloworld.out
```

Output:

```
$ ./helloworld.out
Hello World!
65025
Goodbye World!
```

Standard Input and Output

- `stdio.h` is a large file that contains many function declarations, in fact there are 827 lines of code for this header file alone.

```c
/* Define ISO C stdio on top of C++ iostreams.
Copyright (C) 1991-2024 Free Software Foundation, Inc.
Copyright The GNU Toolchain Authors.
This file is part of the GNU C Library.
...
*/
#ifndef _STDIO_H
#define _STDIO_H        1
...
/* Write formatted output to stdout.
This function is a possible cancellation point and therefore not
marked with __THROW.  */
extern int printf (const char *__restrict __format, ...);
```

https://code.woboq.org/userspace/glibc/libio/stdio.h.html

Primitive Data Types

Primitive Data Types

C has several data types and all variables **must** have a data type

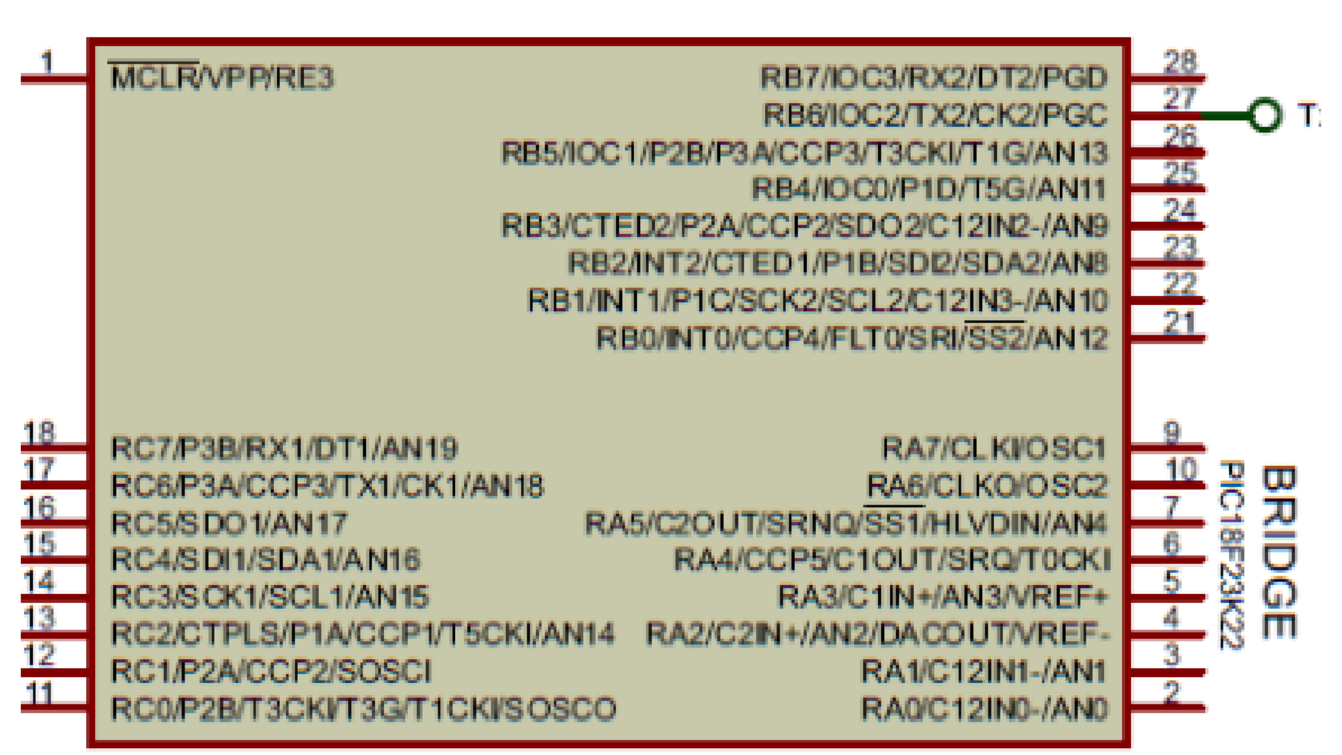
Data Type	Size (Bytes)	Range	Format Specifier
(unsigned)char	at least 1	-128 to 127 or 0 to 255	`%c`
(unsigned)short	at least 2	-32768 to 32767, 0 to 65535	`%h`
(unsigned)int	at least 2	$-2,147,483,648$ to $2,147,483,647$ 0 to 4294967295	`%u`, `%d`
long	least 4	$-9,223,372,036,854,775,808$ to $9,223,372,036,854,775,807$	`%l`, `%ll`, `%lld`, `%lli`
unsigned long	at least 4	0 to $18,446,744,073,709,551,615$	`%lu`, `%llu`
float	at least 2	$3.4e-038$ to $3.4e+038$	`%f`
(unsigned)double	at least 8	$1.7e-308$ to $1.7e+308$	`%lf`
long double	at least 10	$1.7e-4932$ to $1.7e+4932$	`%Lf`

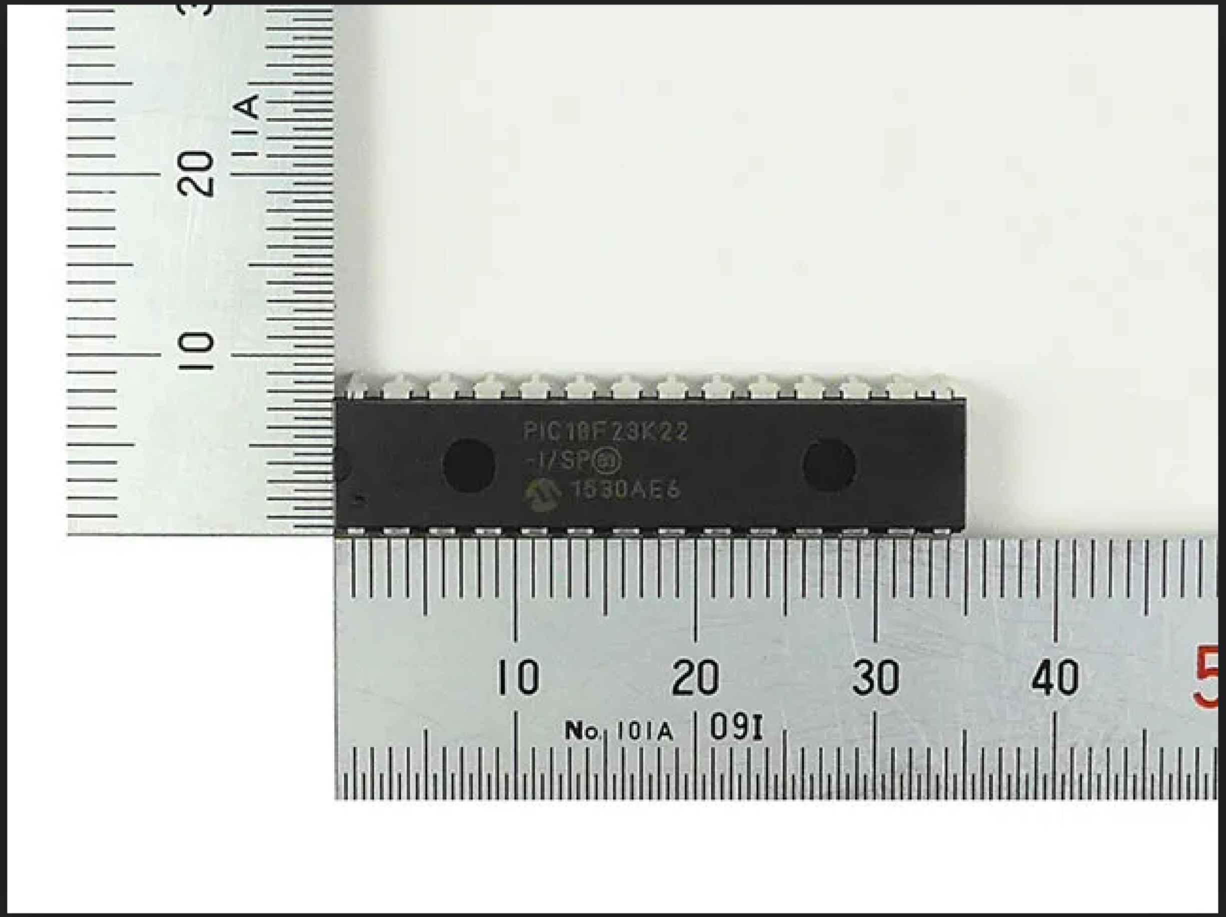
C Advanced Features

- Pointers and addressing, `int*`, `&var1` (more about this later)
- `struct`
  - Allows to combine data items of different kinds
  - `struct Books { char title[50]; char author[50]; int book_id;} book`
- `enum`
  - It consists of constant integrals or integers that are given names by a user.
  - `enum enum_name{int_const1, int_const2, int_const3, …. int_constN};`
- `union`
  - allows to store different data types in the same memory location
  - `union Data { int i; float f; char str[20];} data;`

Embedded C

```
ANSEL  = 0;       // Configure AN pins as digital I/O
ANSELH = 0;
C1ON_bit = 0;     // Disable comparators
C2ON_bit = 0;
//          76543210
TRISC  = 0b10000000;   // PORTC is input
UART1_Init(9600); // Initialize UART PROTO

...

ANSELA  = 0;       // Configure AN pins as digital I/O
ANSELB  = 0;
ANSELC  = 0;
TRISB = 0;
```


1 MCLR/VPP/RE3
28 RB7/IOC3/RX2/DT2/PGD
27 RB6/IOC2/TX2/CK2/PGC
T:
26 RB5/IOC1/P2B/P3A/CCP3/T3CKI/T1G/AN13
25 RB4/IOC0/P1D/T5G/AN11
24 RB3/CTED2/P2A/CCP2/SDO2/C12IN2-/AN9
23 RB2/INT2/CTED1/P1B/SDI2/SDA2/AN8
22 RB1/INT1/P1C/SCK2/SCL2/C12IN3-/AN10
21 RB0/INT0/CCP4/FLT0/SRI/SS2/AN12
18 RC7/P3B/RX1/DT1/AN19
17 RC6/P3A/CCP3/TX1/CK1/AN18
16 RC5/SDO1/AN17
15 RC4/SDI1/SDA1/AN16
14 RC3/SCK1/SCL1/AN15
13 RC2/CTPLS/P1A/CCP1/T5CKI/AN14
12 RC1/P2A/CCP2/SOSCI
11 RC0/P2B/T3CKI/T3G/T1CKI/SOSCO
9 RA7/CLKI/OSC1
10 RA6/CLKO/OSC2
7 RA5/C2OUT/SRNQ/SS1/HLVDIN/AN4
6 RA4/CCP5/C1OUT/SRQ/T0CKI
5 RA3/C1IN+/AN3/VREF+
4 RA2/C2IN+/AN2/DACOUT/VREF-
3 RA1/C12IN1-/AN1
2 RA0/C12IN0-/AN0
BRIDGE
PIC18F23K22



Objective-C

```
#import "MyClass.h"

@implementation MyClass
- (id)initWithString:(NSString *)aName
{
    // code goes here
}

+ (MyClass *)myClassWithString:(NSString *)aName
{
    // code goes here
}
@end
```

Community Experience Distilled

Learning Objective-C by Developing iPhone Games

Leverage Xcode and Objective-C to develop iPhone games

Amy M. Booker
Joseph D. Walters

PACKT

Compilation

Throughout this session we have been using `gcc` or `the GNU Compiler Collection'. The GNU is a recursive acronym: ‘GNU’s Not Unix!'

Supports:

- C,
- embedded-C ,
- Objective-C,
- C++,
- Fortran,
- Ada,
- Go,
- and D

- Example **C** code

```c
int square(int num) {
    return num * num;
}
```

- Example assembley code from `gcc`

```
square:
 push   %rbp
 mov    %rsp,%rbp
 mov    %edi,-0x4(%rbp)
 mov    -0x4(%rbp),%eax
 imul   %eax,%eax
 pop    %rbp
 ret
```

Command Line Arguments

```
$ mkdir arguments && cd arguments
$ touch arguments.c
$ vim arguments.c
```

- `main()` can now take an integer as an argument.

```c
#include <stdio.h>

int main( int argc, char *argv[] )  {

   printf("Program name %s\n", argv[0]);

   if( argc == 2 ) {
      printf("The argument supplied is %s\n", argv[1]);
   }
   else if( argc > 2 ) {
      printf("Too many arguments supplied.\n");
   }
   else {
      printf("One argument expected.\n");
   }
   return 0;
}
```

- `argv[0]` this is an array, at index 0 is the current programs file name... **always**.

```c
#include <stdio.h>

int main( int argc, char *argv[] )  {

   printf("Program name %s\n", argv[0]);

   if( argc == 2 ) {
      printf("The argument supplied is %s\n", argv[1]);
   }
   else if( argc > 2 ) {
      printf("Too many arguments supplied.\n");
   }
   else {
      printf("One argument expected.\n");
   }
   return 0;
}
```

- `if( argc == 2 )` checks to see if the number of arguments supplied is 2, (program name is 1)

```
#include <stdio.h>

int main( int argc, char *argv[] )  {

   printf("Program name %s\n", argv[0]);

   if( argc == 2 ) {
      printf("The argument supplied is %s\n", argv[1]);
   }
   else if( argc > 2 ) {
      printf("Too many arguments supplied.\n");
   }
   else {
      printf("One argument expected.\n");
   }
   return 0;
}
```

- `...%s\n", argv[1])` gets the argument you supplied and then formats it as string to the terminal

```c
#include <stdio.h>

int main( int argc, char *argv[] )  {

   printf("Program name %s\n", argv[0]);

   if( argc == 2 ) {
      printf("The argument supplied is %s\n", argv[1]);
   }
   else if( argc > 2 ) {
      printf("Too many arguments supplied.\n");
   }
   else {
      printf("One argument expected.\n");
   }
   return 0;
}
```

- `else if( argc > 2 )` if the first `if` is `false`, then check to see if you have supplied 2 arguments

```c
#include <stdio.h>

int main( int argc, char *argv[] )  {

   printf("Program name %s\n", argv[0]);

   if( argc == 2 ) {
      printf("The argument supplied is %s\n", argv[1]);
   }
   else if( argc > 2 ) {
      printf("Too many arguments supplied.\n");
   }
   else {
      printf("One argument expected.\n");
   }
   return 0;
}
```

- `else` if you have not supplied an argument

```c
#include <stdio.h>

int main( int argc, char *argv[] )  {

   printf("Program name %s\n", argv[0]);

   if( argc == 2 ) {
      printf("The argument supplied is %s\n", argv[1]);
   }
   else if( argc > 2 ) {
      printf("Too many arguments supplied.\n");
   }
   else {
      printf("One argument expected.\n");
   }
   return 0;
}
```