Review of Technical Background

The Nature of the Interfacing Problem

- Physical laws do not change
  => Currents & voltages obey the laws of Kirchoff and Ohm.
  => Signal propagation and noise calculations are the same.
  => But digital & software systems don't obey simple laws!

- Hardware (H/W) is constantly evolving because of technological improvements and inter-company competition.
  => System size & complexity increase rapidly with time.
  => New features get added; old features are often retained.
  => Multiple versions of the hardware must often co-exist.

- Software (S/W) flexibility is both an advantage and a danger.
  => Complex H/W must be initialized & configured in S/W.
  => S/W system complexity overwhelms any one designer.
  => Multiple versions of software must often be maintained.
  => Software invariably contains design defects ("bugs").

Typical Mechanisms at Hardware Interfaces

1) *Analog signal conditioning*
e.g. buffers/drivers, noise rejection filters, anti-aliasing filters

2) *Analog-to-digital* and *digital-to-analog* signal conversion
e.g. Analog inputs from *sensors* need to be converted into digital form to permit digital signal processing (DSP).
e.g. Digital outputs need to be converted into voltage or current signals to control *actuators* in the analog world.

3) *Digital modulation* of an analog signal
e.g. Digital signals must be encoded as modulated analog signals to propagate well over communication channels.
Modem = modulator (transmitter) + demodulator (receiver)

4) *Timing recovery*, *demodulation* & *decoding* a modulated signal. Present in digital communication receivers.

Typical Mechanisms at Hardware Interfaces

5) Digital signal *conversion*, *conditioning* and *isolation*
  e.g. Voltage level conversion, voltage-current conversion
  e.g. Optical isolation transceivers, isolation transformers

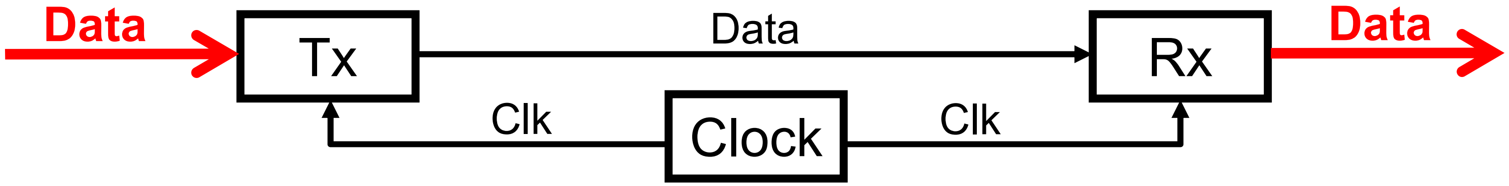
6) *Impedance matching (to avoid unwanted signal "ringing")*
  e.g. Parallel termination resistance at the far end of a line
  e.g. Series termination resistance at the near end of a line

7) *Synchronization*, *framing*, *handshaking*, *error detection*
  e.g. Start and stop bits in RS-232C
  e.g. Request-to-send and clear-to-send handshaking
  e.g. Parity bit (or checksum) generation at transmitter
  e.g. Parity bit (or checksum) re-generation and checking at receiver. Possibly error correction at receiver.

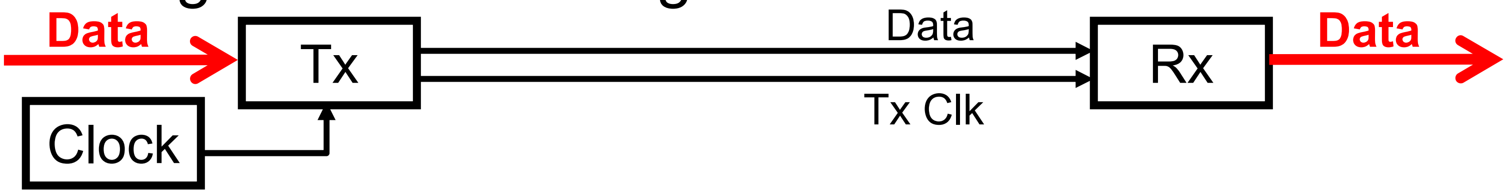
8) *Data buffering* and *flow control* to handle rate fluctuations
  e.g. Hardware buffers inside transmitters and receivers

Synchronous Digital Interfaces

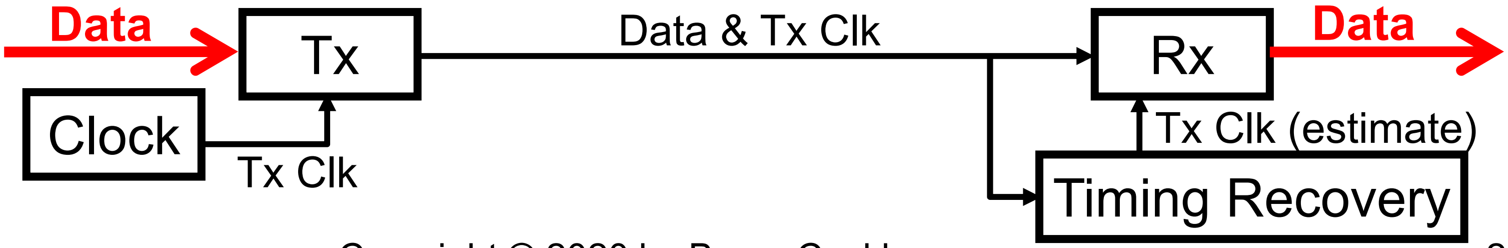
1) Transmitter (Tx) and receiver (Rx) clock signals are both directly derived from the same physical clock signal.



2) Transmitter clock is sent to the receiver as a separate signal along with the data signal.



3) Receiver clock is synchronized with a clock signal that is encoded along with the transmitter's data signal. There is no separately transmitted transmitter clock signal.



Asynchronous Digital Interfaces

- The transmitter and receiver sides use physically distinct and unrelated clock signals. The clock frequencies may be nominally the same. More commonly, the raw receiver clock is some multiple (say 16x) of the transmitter clock.

- For example, if the receiver clock is 16x the transmitter clock, then the raw receiver clock can be divided by 16, and given one of 16 possible phases that is (at least initially) closest to the estimated phase of the transmitter clock.

- Thus the original transmitter clock can be approximated by the divided-down receiver clock. But this approximate clock will drift away in phase due to any frequency offset between the transmitter and divided-down receiver clocks.

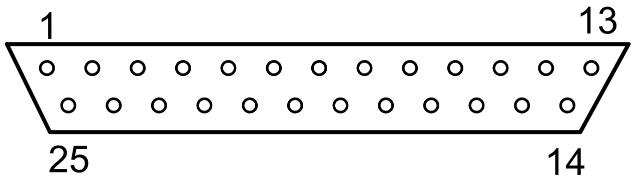
Semisynchronous Digital Interfaces

- Like in a synchronous interface, there is a common clock.
- Time is measured out as a number of whole clock periods.
- Communication events (e.g., read & write operations on a bus) take a *variable number of clock periods* to complete.
- The number of clock periods can be determined either by (1) handshake signals "on-the-fly" at the time of the event, or (2) by some fixed number that is programmed in the hardware.
- Hence we get much of the *timing flexibility* of an asynchronous interface, while retaining many of the *simplifications of synchronous (i.e., clocked) design*.
- Most modern digital interfaces are semisynchronous.

Serial Hardware Interfaces

- An interface through which the *data bits* are transmitted *one bit at a time* in each direction.

Ex. RS-232C



Pin 1: ground (GND)

Pin 2: *transmit data* (TxD)

Pin 3: *receive data* (RxD)

Pin 4: request to send (RTS)

Pin 5: clear to send (CTS)

Pin 6: data set ready (DSR)

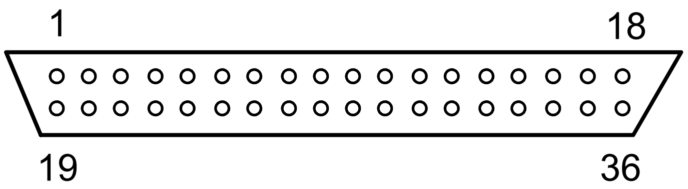
Pin 20: data terminal ready (DTR)

- *Note*: A smaller 9-pin version was widely used in PCs.

Parallel Hardware Interfaces

- An interface through which *multiple data bits* are transmitted *at the same time, in parallel*, over different connections.

Ex. Centronics printer I/F



Pin 1: data strobe

Pins 2-9: *8 data bits*

Pin 10: acknowledge

Pin 11: busy

Pin 12: paper end

Pin 13: select

Pin 17: ground

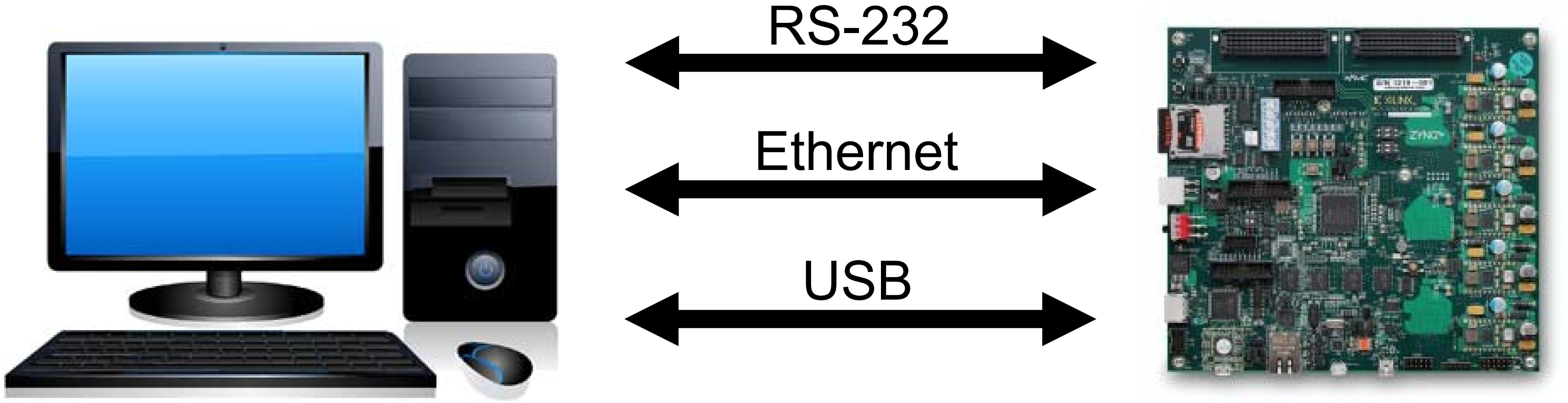
Pin 18: +5 V

Pin 32: fault detected

Examples of Hardware Interfaces

	Synchronous	Asynchronous	Semisynchronous
Serial	USB peripheral	RS-232C terminal	ATM telecom
Parallel	6800 µP bus	GPIB instrument	ColdFire bus (ECE 315 system)

Embedded Software Development (1)



RS-232

Ethernet

USB

Host system contains:

- full-featured O.S.
- Internet connection
- IDE
- source version control
- (usually) a cross compiler

Target system contains:

- boot loader
- ASCII interface
- monitor for debugging
- (often) file system
- (rarely) native compiler

Embedded Software Development (2)

- Embedded systems range widely in performance and features (e.g., simple 4-bit microcontrollers, complex 32-bit microcomputers, microprocessor + FPGA on one chip).
- The variety and complexity (both hardware and software) of embedded systems tends to make embedded software development a challenging and hence expensive task.
- A variety of different software tools must be used during embedded software development: source code editors, compilers, build automation tools, debuggers, monitors, etc.
- Unless the tools are designed in a consistent way, each tool will tend to have a different user interface. Going from tool to tool will thus require mental “mode switching”, which is tiresome leading to lower programmer productivity and bugs.

Integrated Development Environments (IDEs)

- An ***integrated development environment*** *(IDE)* is a single application that executes on the host system (e.g., personal computer or workstation) where software is being designed.
- A typical IDE (e.g., CodeWarrior, Eclipse) provides a complete set of software development tools (e.g., source code editor, compiler, build automation tool, version control system, debugger, etc.) that have a ***consistent user interface*** in order to maximize programmer productivity and to increase the quality (e.g., reduce the number of bugs, increase the maintainability) of the resulting software.
- The complexity of an IDE can require a considerable initial learning effort. However, increased productivity should result once the “learning curve” has been climbed.

Automated Version Control Systems (1)

- Both software and hardware design activities generate a large number of computer files containing design details.
- In general, humans are terrible at keeping track of a large number of details. Also, misunderstandings among the members of a design team can easily lead to design errors.
- Only certain versions of different design files will correspond to consistent (and potentially working) versions of the entire system that were developed at one particular time.
- A ***version (or revision) control system*** is a computerized file library (or repository) that keeps track of the design files such as documentation, source files for software, source files for hardware, configuration files, test plans and scripts, design bug reports, engineering change orders (ECOs), etc.

Automated Version Control Systems (2)

- There are many available version control systems:
  - RCS, CVS, Perforce, Subversion, Git, *etc*.
- These systems keep track of the same kinds of information:
  - Name, version number and date for all versions of a file
  - When, why and by whom each file was first created
  - Designer name and reasons for each version of a file
  - The current version of all files
  - Sets of consistent versions of design files
  - File status: *active* and up-to-date, *checked-out* by one (and possibly more) designers; *frozen*; *obsolete*
  - When and by whom a file was "checked out" for change
  - *Etc*.

Automated Version Control Systems (3)

- Typical commands in version control systems:
  - *create* a new file with a given name
  - view the version (or revision) *history* of a file
  - obtain a copy of a *specific version* of a file
  - *check out* (and possibly *lock*) a file for modifications
  - *identify the designers* who have checked out a file
  - *check in* (and possibly *unlock*) one modified file
  - *check in* a consistent related group of modified files
  - *merge* multiple checked-out files into one file (tricky!)
  - *release* a stable and consistent set of design files
  - *roll back* to a previous consistent system design version
  - *Etc*.

Assembly Language vs. Compiled Code

- There was a long tradition of using assembly language programming in embedded systems. This was done in the belief that compilers produced inefficient code, and that an expert programmer could take better advantage of the features of the CPU-specific machine language.
- However, assembly language code is rarely used today.
  - o Compiler technology is much better today.
  - o Programmer productivity (even for experts) is much too low in assembly language compared to compiled code.
- Assembly language might be seen today in a few places, e.g.
  - o Context switch and other interrupt service routines
  - o High-performance graphics and signal processing S/W

Selection of a Programming Language

- Many factors influence the choice of programming language:
  - ❖ Language features: enhanced productivity & safety
  - ❖ Higher-level languages usually increase both human productivity and system safety
  - ❖ Compatibility with the existing investment in software
  - ❖ Compiler & software support for the hardware platforms
  - ❖ Availability of kernels, hardware drivers, TCP/IP stacks
  - ❖ The preferences & experience of the S/W designers
- *There is no perfect programming language*. Other factors (e.g., the software development processes and practices in actual use) typically play the most important role in determining the success of a software development project.

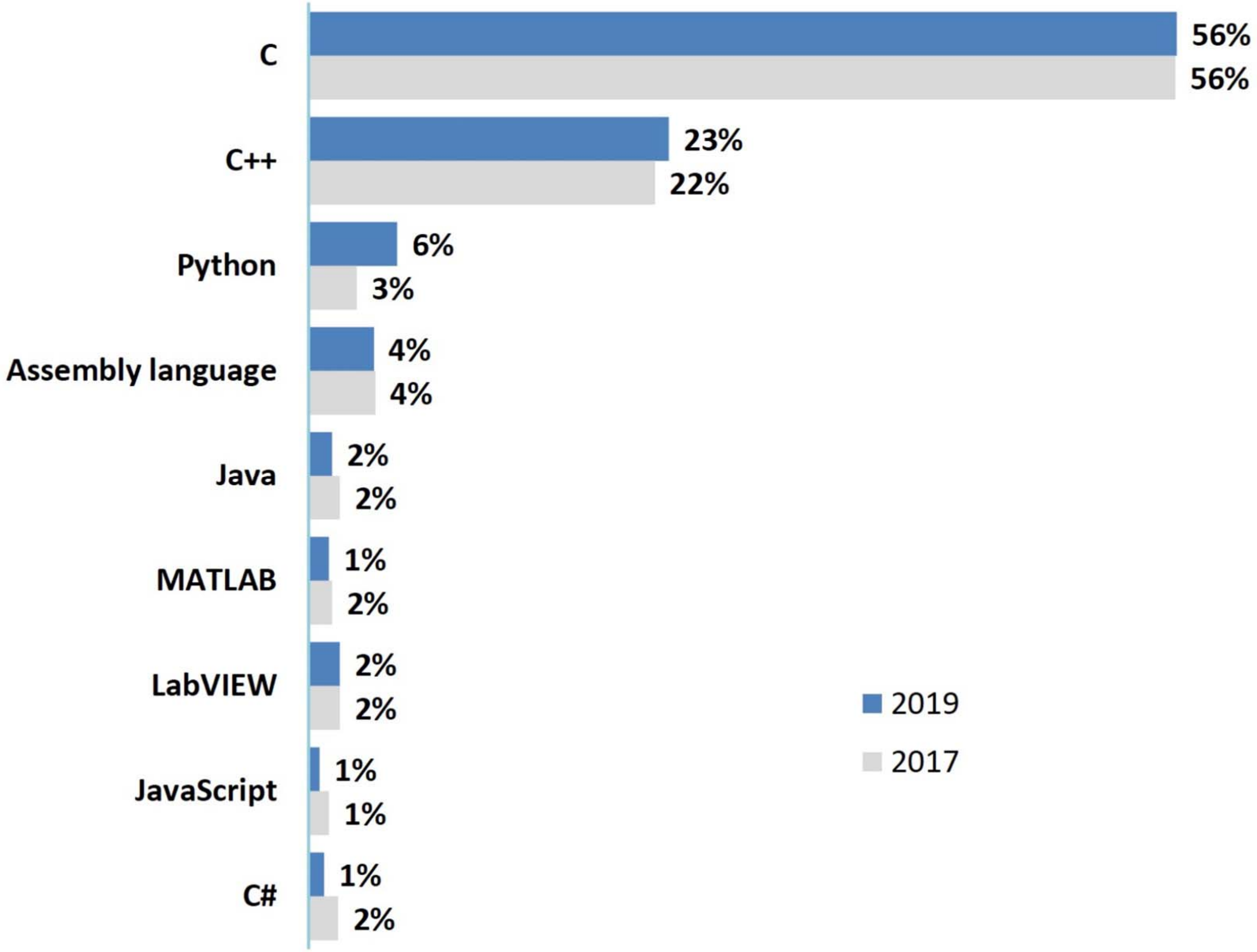
Software Engineering

- "*Software engineering is the application of engineering to the design development, implementation, testing and maintenance of software in a systematic method*." [Wikipedia]
- There is no one best software engineering methodology, but some proven systematic approach must be adopted in the development of software. The risks and costs of problems in undisciplined software development are too great to ignore.
- At a minimum, there should be documentation to cover the software requirements and specifications, coding standards, commenting standards, design reviews for software changes, software defect (bug) reporting and resolution, etc.
- Use the features in the language (e.g., the freedom to name identifiers and to include meaningful comments) to make the code self-documenting or at least easier to understand.

The C Programming Language

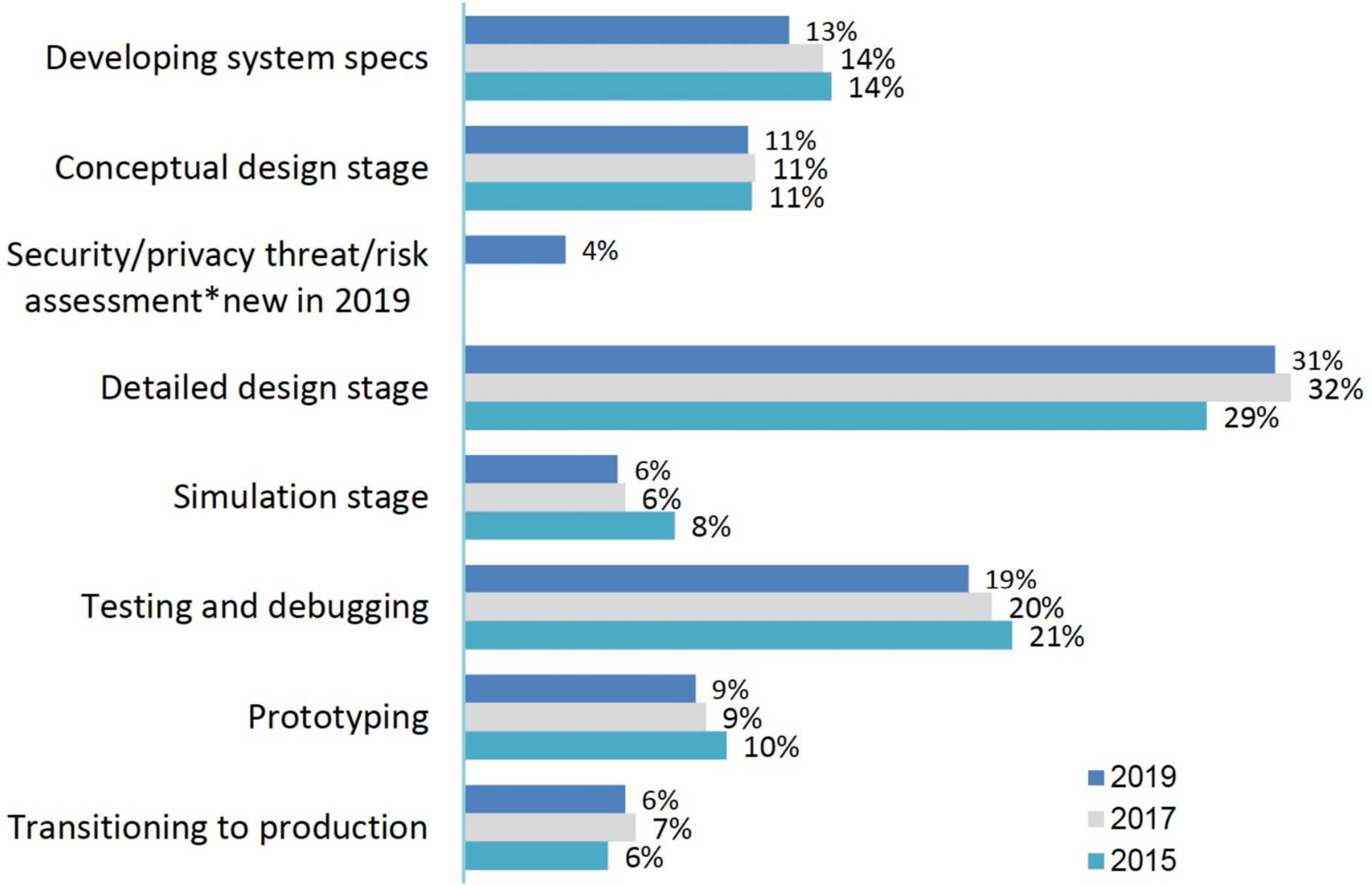
- The C programming language is one of the most widely used languages. Most computer architectures have C compilers.
- Developed by Dennis Ritchie at Bell Laboratories over 1969-1973, C was intended to provide both high-level language constructs (e.g., loops, pointers, I/O) as well as fairly direct access to memory structures, such as words, bytes and bits.
- The flexibility of C is accompanied by many potential hazards.
- Historically, embedded systems have migrated from assembly language to the widespread use of C as the best compiled (but still relatively low-level) language alternative.
- C’s dominant position in embedded systems is likely to diminish over time (there are safer languages available now) but the existing C code base will remain very large.

The Top Languages for Embedded Systems



Courtesy of AspenCore’s 2019 Embedded Markets Study

Time Spent on the Design Stages



Courtesy of AspenCore's 2019 Embedded Markets Study

Some Pros and Cons of C

Pros

- Widely supported across microcomputer architectures
- Used to implement the popular Unix/Linux operating syst.
- The largest existing code base for embedded systems

Cons

- C missed out in progress in language design since 1970
- C syntax is compact and “powerful”, but also hazardous
- Poor support for data encapsulation & information hiding
- Haphazard rules for exporting symbol links across files
- Dynamic memory allocation and recycling can be tricky
- Pointers are the source for many serious design errors . . .

Pointers in C

- A ***pointer*** is a variable that stores the ***base address*** of data (e.g., variables, arrays, strings, data structures, software “objects”) or a piece of code (e.g., functions) in memory.
- The flexibility of pointers is useful for forming & accessing data structures, and for passing complex parameters. However, *pointer errors can easily lead to serious bugs and crashes.*
- Pointers are declared much like all other variables. Pointers can be typed; however, the type can be effectively overridden.

```
int var1;    /* variable var1 has type int */
int *Ptr1;   /* pointer Ptr1 has type (int *) */
int * Ptr1;  /* same as previous line */
char *s;     /* s has type (char *) */
int * Ptr1, Ptr2; /* Ptr1 is ptr of type (int *) */
                  /* Ptr2 is var of type int */
int *Ptr1, *Ptr2;  /* two ptrs of type (int *) */
void *Ptr1, *Ptr2; /* two typeless pointers */
```

Creating Pointer Values in C

- The value of a newly declared variable pointer is not defined. *Many program bugs are caused by attempts to use pointers that have not yet been assigned valid pointer values.*
- The predefined constant **NULL** is a pointer value that points to nothing. In C++, the value 0 is often used instead of **NULL**.
- The ***reference operator*** & extracts the ***base address*** of any given software structure (e.g., function, variable, pointer).

```
int var1, *Ptr1, **Ptr2;
Ptr1 = &var1;     /* Ptr1 points to var1 */
Ptr2 = &Ptr1;     /* Ptr2 points to Ptr1 */
```

- Pointer values can also be derived from other pointers, or they can be returned as output values from functions.

```
int *Ptr1, *Ptr3;
Ptr1 = Ptr2+1; /* Ptr2 must hold an (int *) value */
Ptr3 = functionThatReturnsIntPointer( 1, 2, var1 );
```

Dereferencing Pointers in C

- The ***dereferencing operator*** * is used to access the data or software code that is being pointed to by a pointer. Ex:

```
int var1, var2, *Ptr;  /* create 2 vars & one ptr */
var1 = 6;              /* initialize var1 */
Ptr = &var2;           /* Ptr points to var2 */
*Ptr = 3;               /* load var2 with value 3 */
var2 = var1 + *Ptr;     /* var2 now holds value 9 */
void (* Fptr)( char, int); /* create function ptr */
Fptr = &NameOfFunction1;    /* use the & operator */
*Fptr( 'g', 3 );          /* execute the function */
Fptr = NameOfFunction1;   /* this also works in C */
*Fptr( 'h', 6 );          /* execute the function */
Fptr = NULL; /* NULL should never be dereferenced */
*Fptr( 'k', 7 );  /* will crash the task or worse */
```

Arrays in C

- Arrays group variables or data structures of the same type.
- Arrays are accessed (inside compiled code) using pointers. *Nothing checks if the pointer goes outside the array limits.*

```
int *Ptr;
int ar[100];          /* create array of 100 ints */
for (i = 0; i < 100; i++)
    ar[i] = 2 * i;     /* load next array element */
Ptr = &ar[0];     /* point to first array element */
for (i = 0; i < 100; i++) {
    Ptr = &ar[i]   /* point to next array element */
    *Ptr = 2 * i;      /* load next array element */
}
Ptr = ar;  /* Also legal: point to 1st array elem */
for (i = 0; i < 100; i++) *(Ptr++) = 2 * i;
```

Example using Pointers

```c
#include <stdio.h>

int *functionThatReturnsIntPointer(int *aPtr, int *bPtr) {
        if (*aPtr > *bPtr)
               return aPtr;
        else if (*aPtr < *bPtr)
               return bPtr;
        else
               return NULL;
}

int main() {
        /* Best to declare all variables at the start */
        int var1[4] = {4, 5, 6, 7};
        int *Ptr1, **Ptr2, *Ptr3, *Ptr4;

        Ptr1 = &var1[0];    /* Ptr1 = var1; also works */
        Ptr2 = &Ptr1;
        Ptr3 = Ptr1 + 2;    /* inside var1 array */
        Ptr4 = functionThatReturnsIntPointer(Ptr1, Ptr3);

        return 0;
}
```

Pointer Arithmetic and Operators in C

- A typed pointer can be manipulated arithmetically.
  - can *add* or *subtract integers* to/from a pointer
  - can *subtract* (but not add) two pointers of the same type
  - can *compare* two pointers of the same type for equality (e.g., **`Ptr1 == Ptr2`** and **`Ptr1 != NULL`**).
- The addition (+) and subtraction (-) operators are modified for pointers to advance the pointer forward or backward along an array of identically typed variables or data structures. The integer value corresponds to the number of array elements. For example, **`*(Ptr1+1) = *(Ptr2–3);`**
- Pointers can be incremented *after* they are used (e.g., **`Ptr2++`**) as well as *before* they are used (e.g., **`++Ptr1`**). Pointers can also be decremented before or after they are used. For example: **`*(--Ptr1) = *(Ptr2--);`**

Character Arrays and Strings in C

- We can create an array of characters (**`char`**’s).
- A *string* is an array of **`char`**’s that is terminated with the ASCII null character ‘**`\0`**’. The terminating null character is not displayed when a string is sent to an output device. It is used by string processing functions so that they can recognize the end of the string without having to be informed of its length.
- The compiler automatically adds the terminating character to *string constants*. For example,

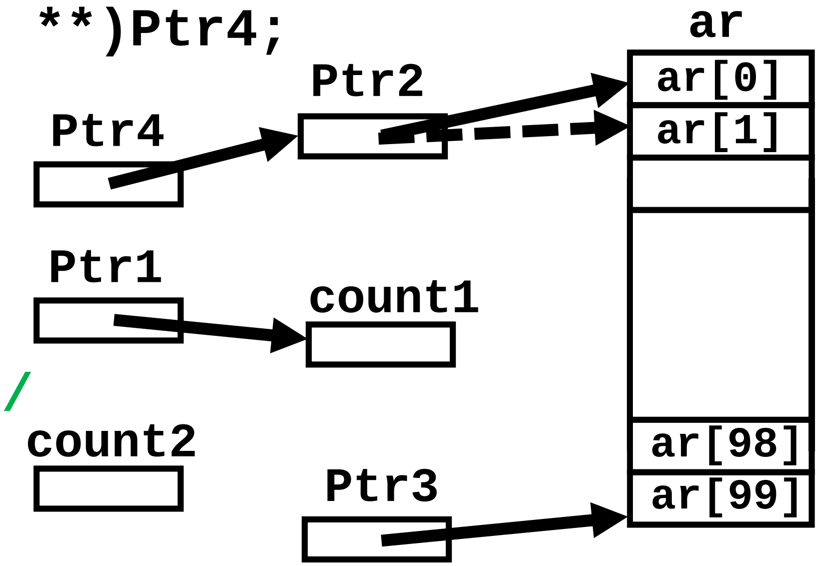
```
printf("hello, " );      /* string has 8 char’s */
printf("world\n");       /* string has 7 char’s */
```

- String input functions automatically append the **`\0`** terminator.

```
char studentName [50];   /* create char buffer */
scanf("%s", studentName );   /* 49 chars maximum */
```

Examples that Use Pointers in C

```
int *Ptr1, *Ptr2, *Ptr3;   /* type is (int *) */
void *Ptr4;             /* same as void * ptr4 */
int count1, count2, ar[100];
Ptr1 = &count1;
Ptr2 = ar;            /* same as Ptr2 = &ar[0] */
Ptr3 = &ar[99];
Ptr4 = (void *)&Ptr2; /* Ptr4 contains addr of Ptr2 */
count1 = 1024;
*(Ptr2++) = 512;
*Ptr3 = *Ptr1 + ar[0] + **(int **)Ptr4;
/*  *Ptr1 contains 1024 */
/*  ar[0] contains 512  */
/*  (int **)Ptr4 refs Ptr2 */
/*  Ptr2 now refs ar[1] */
/*  ar[1] has unknown value */
/* **(int **)Ptr4 is unknown */
/* ar[99] contains 1536 */
/*  if ar[1] contained 0 */
```



Casting Pointers in C (1)

- Typed variables & pointers, together with run-time type checking, provide a powerful way of avoiding or detecting software design errors.
- However, to obtain greater efficiency, flexibility and portability (with some loss in safety) it is common for embedded system software to override type checking, especially for pointers.
- The C programming language allows the types of variables and pointers to be *cast* into a more general type.
- To avoid type checking, pointer values to typed objects can be assigned to a pointer that is of type **`(void *)`**.
- General functions can be designed to manipulate**`(void *)`**'s. However, a **`(void *)`**cannot be dereferenced. We must first cast the **`(void *)`** to a pointer to a typed object (e.g., an **`int *`**) and then dereference it to obtain the object (e.g., an **`int`**).

Casting Pointers in C (2)

```
#include <stdio.h>

void printAnInteger(void *argPtr) {
        printf("Integer = %d", *(int *)argPtr);
}

void printAString(void *argPtr) {
        printf("Character = %s", (char *)argPtr);
}

int main() {
        int var1 = 2017;
        char str2[] = "2017";
        void *argPtr;
        void (* funPtr)(void *)

        funPtr = printAnInteger;
        argPtr = (void *)&var1;
        *funPtr( argPtr );   /* typeless argument is passed */

        funPptr = printAString;
        argPtr = (void *)str2;
        *funPtr( argPtr );   /* typeless argument is passed */

        return 0;
}
```

Some Common Dangers in C Code

- ***Using a pointer*** variable that has ***not yet been initialized*** or has been ***corrupted***. This problem typically leads to bus errors or segmentation faults followed by system crashes.
- ***Buffer overflow***: Adding elements into an buffer (implemented as an array) beyond the buffer's capacity. This can cause system crashes as well as security breaches. This problem can occur using standard functions, like **`strcpy`** and **`gets`**, which rely on the presence of a terminating **`\0`** or **`EOF`**. Use **`strncpy`** and **`fgets`** instead.
- ***Type checking can be easily overridden*** in C using casting and **`void`** pointers. For example, a 32-bit **`double`** integer can be stored at one address and then read out and (only the upper byte) used as an 8-bit **`char`**.

Software Objects in C++ (1)

- Programming languages are provided with features that increase programmer productivity and that avoid errors.
- A software ***object*** is a data structure that gathers together *private* and/or *public state variables* and the functions (also called *member functions* or *methods*) that must be used when operating on private variables.
- Software objects are often a convenient way of modelling the behaviour of physical objects, hardware subsystems, complex data structures, peripheral interfaces, etc.
- Private variables and implementation details can be safely hidden using software objects.
- New object instances are initialized in a consistent way.
- Unneeded objects are recycled in a consistent way.

Software Objects in C++ (2)

- Objects are instantiated from ***classes*** that define the state variable types and functions.
- Classes can be organized into inheritance hierarchies so that commonalities and differences among the classes can be clearly expressed and possibly exploited.
- Each object instantiation has its own separate set of instances of the state variables.
- An object ***constructor function*** is called automatically when an object is created and initialized. This guarantees that object initialization is done consistently and safely.
- An object ***copy constructor function*** is called automatically when an object is passed into a function as a parameter. The C.C.F. ensures that the passed copy has its own dynamically created subobjects & pointers.
- A ***destructor function*** is called automatically when an object is deleted. This ensures that any resources attached to the object (e.g., dynamically allocated memory) are safely recycled.

Software Objects in C++ (3)

```
class Register8 {
public:
   Register8();      // constructor function
   // Copy constructor and destructor fcns not used here.
   //Register8( const Register8 &obj );  // A copy
   // constructor is required if the class objects contain
   // pointer variables and dynamically allocated objects.
   //~Register8();   // destructor not used in this class
   void setBit( int );
   void clearBit( int );
   void enableOutput();
   void disableOutput();
private:             // here all vars are private
   bool register[8];
   bool outputEnable;
};

Register8::Register8() {  // constructor function
   outputEnable = 0;      // init all private vars
   for (i=0; i<8; i++) register[i] = 0;
}
```

Software Objects in C++ (4)

```
void Register8::setBit( int b ) {
   if ((b > -1 ) && (b < 8)) register[b] = 1;
}

void Register8::clearBit( int b ) {
   if ((b > -1 ) && (b < 8)) register[b] = 0;
}

int Register8::getBit( int b ) {
   if ((b > -1) && (b < 8))
      return register[b];
   else
      return -1;         // error return code
}

void Register8::enableOutput() {
   outputEnable = 1;
}

void Register8::disableOutput() {
   outputEnable = 0;
}
```

Software Objects in C++ (5)

```
int main()
{
   Register8 redReg;         // Create 3 Register8
   Register8 yellowReg;      // objects.  They will
   Register8 greenReg;       // be initialized.

   redReg.setBit(2);         // Operate on redReg
   redReg.setBit(4);
   redReg.enableOutput();

   yellowReg.setBit(7);      // Operate on yellowReg
   yellowReg.enableOutput();

   if ( redReg.getbit(4) )   // should check for -1
      greenReg.setBit(5);
   else
      greenReg.setBit(7);
   greenReg.enableOutput();

   return 0;
}
```

Global vs. Local Variables

- A ***global variable*** is a variable that is accessible by any function, unless the variable is “masked” by a local variable with the same name.
- A ***local variable*** is a variable that is defined within the scope of the **`main()`** function or any other function.

```
int sysTime;  /* Global variable declaration */
/* sysTime is readable by all functions unless masked */
int main()
{
     int addTime( int x );  /* Local function prototype */
     /* Additional lines of code can appear here */

     int addTime( int x )  /* Local function definition */
     {
          int y;     /* Local variable declaration */
          y = x + sysTime;  /* Can access a glob variable */
          return y;
     }               /* Variable y is now recycled */
     return 0;     /* Return value 0 means no error */
}
```

Global Variables Can Be Dangerous

- All functions can access a global variable, which is both a convenience and a potential hazard.
- The convenience is that global variables can be accessed very fast, with no overhead due to parameter passing.
- *The main hazard of global variables is that they are a path for complicated and potentially unwanted side-effects.* The hazards of such interactions are even harder to predict and understand in multitasking systems (described later).
- Duplicate global variable names can arise when merging software projects, which can easily lead to errors.
- In general, ***avoid using global variables*** if at all possible.
- The only safe situation for using a global variable is to store a global parameter that is declared in a well-documented header file, and only possibly read by other routines. Note that C does not prevent multiple writers to a global variable.

Memory Allocation for Objects in C

- Variables, data structures and other objects require memory space, which must be allocated and managed during the lifetime of the software system.
- C takes an old-fashioned, low-level approach that gives more direct control (and responsibility) to the programmer.
- Most modern programming languages (e.g., Java) avoid “raw pointers” and explicit dynamic memory allocation.
- ***Static memory allocation***: Memory space is allocated to the object for the lifetime of the software system.
- ***Automatic memory allocation***: Memory space is allocated automatically when the object is instantiated and de-allocated when the context of the object is exited.
- ***Dynamic memory allocation***: Memory space is requested as required from the operating system. Allocated memory must be returned to the O.S. when it is no longer required.

Static Memory Allocation in C

- ***Static memory allocation***: Memory is allocated to the object for the lifetime of the software system.
- Static memory allocation is handled by the compiler tool chain. Sufficient memory space is assigned in a suitable region of the memory map (e.g., near the program code).
- Typical examples:
  - **External objects**, which are declared outside of all the functions. These are visible across all source files.
  - An **external object** that is also declared to be **`static`** is unknown outside of the one source file.

    ```
    static int temperature;
    ```
  - An **internal object**, which is one that is defined inside a function, that is declared to be **`static`**. The object's value (or state) is preserved across function calls.

Register Variables in C

- The registers in the central processing unit (CPU) are normally used to hold intermediate values or variables that have very limited scope.
- CPU registers offer the fastest possible implementation for a variable, but the number of registers is quite limited.
- To create fast code, modern compilers attempt to implement as many variables as possible as CPU registers.
- In some applications, a small number of variables might be very heavily used. In such cases, the programmer may wish to force the use of CPU registers for some variables to obtain the greatest possible performance.
- The register keyword can be used in C programs as a *strong hint* to the compiler to use a register for a variable.

```
register long totalcount;
```

Automatic Memory Allocation in C

- **Automatic memory allocation**: Memory is allocated automatically when a new object is instantiated inside a context, and de-allocated later when the context is exited.
- Implemented using a hardware-supported stack. All modern CPUs provide one or more stacks for this purpose.
- Typical examples:
    - *Function arguments* (passed by value into a local copy)
    - *Local* or *internal objects* declared inside functions
    - *Local objects* declared inside brace-delineated blocks
- ***Stack overflow*** is the failure when there is an attempt to store more data in a stack than the stack has room for.
- Common causes: (1) infinite recursion, (2) storing an overly large object or too many objects onto the stack.

Dynamic Memory Allocation in C

- **Dynamic memory allocation**: Memory is requested as required from the operating system. Allocated memory must be returned to the O.S. when it is no longer required.
- The ***heap*** is a region of memory that is used by the O.S. to provide space to implement all of the dynamic objects.
- **`malloc( N )`** allocates **`N`** > 0 bytes from the heap and returns a pointer to the base addr. of that region in memory.

```c
int *array = (int *)malloc( 20 * sizeof(int) );
int *array = malloc( 20 * sizeof(int) );  // also
// Note! malloc returns a (void *) if the
// bytes are allocated; otherwise, NULL
```

- Use `free( ptr )` to recycle a dynamic object back to the heap

```c
free( array );
free( &array );     // same effect
free( &array[0] );  // same effect
```

Potential Hazards of Dynamic Memory (1)

- Dynamic memory allocation in C assumes that the programmer will avoid errors during both the allocation and de-allocation steps. Such errors can have serious effects.
- ***Using an uninitialized pointer variable*:** A pointer variable must be properly initialized to point to an object with allocated memory; otherwise, serious system failures will likely occur as a result of the subsequent bus error or segmentation fault (or the overwriting of data or code).
- ***Failure to check for allocation failure*:** When **`malloc()`** fails to be allocated all of the requested bytes, a NULL pointer is returned. This possibility must be checked for and dealt with safely or else the program will go on to use a NULL pointer value and likely cause a bus error or segmentation fault, very likely leading to a system crash.

Potential Hazards of Dynamic Memory (2)

- **Failure to call free() to recycle dynamic memory:** If a dynamic object is not freed after the object is no longer required, the memory space allocated to that object is effectively removed from the heap. This situation is called a **memory leak**. Over time memory leaks will degrade system performance and eventually cause failure.
- **Calling free() more than once for the same pointer value:** This could cause a new object to be corrupted.
- **Using a pointer to a dynamic object that has already been freed:** This is called a *dangling pointer* bug. Be very careful when a pointer to a dynamic object is assigned to other pointer variables! Freeing the object using one of the pointers will invalidate the pointer value in all of the other pointers that still point to the (now recycled) dynamic object.

Dynamic Memory Allocation in C++

- C++ introduces new functions that offer safer and more convenient dynamic memory allocation than those in C.
- The **new** function returns a pointer to memory from the heap for a new instance of the object class and also automatically calls the object's constructor function (if one exists).

```
        TypeName *typeNamePtr;
        typeNamePtr = new TypeName;
```

- The **delete** function calls the object's destructor function (if one exists) and then deallocates the memory to the heap.

```
        delete typeNamePtr;
```

- The **new []** and **delete []** functions must be used to allocate and deallocate an array of objects.

```
        int *intArrayPtr;
        intArrayPtr = new int [100];
        delete [] intArrayPtr;
```

Garbage Collection

- **Garbage collection** is a mechanism that is used in many software environments (e.g., Java, C#, Matlab, Python, but not C or C++) to automatically reclaim *garbage memory*, that is, memory space that is used to implement objects that are no longer in use and no longer required.
- Garbage collection is intended to eliminate potentially serious problems like memory leaks, dangling pointer bugs, double free bugs, etc. It can also be integrated along with algorithms that reduce memory fragmentation.
- However, garbage collection is challenging to implement in a way that does not impact system performance. This is the reason why garbage collectors are uncommon in embedded systems, especially real-time embedded systems.
- Garbage collection can co-exist in a system that also provides manual dynamic memory allocation.

Smart Pointers in C++

- C++ does not have built-in garbage collection.
- Also, the `new` and `delete` functions still leave open the possibility of serious pointer-related problems
- A ***smart pointer*** is an abstract data type, introduced in C++98, that provides pointer functionality with enhancements that aim to avoid serious errors like initialization errors, dangling pointers, attempting to move a pointer to beyond its valid address range, etc.
- A smart counter can use a ***reference counter*** to determine the number of users of a dynamically allocated region of memory. Only after the reference count goes to zero will the memory be released back to the heap using a destructor function.
- Different smart pointers were defined in C++98, 11 & 17.