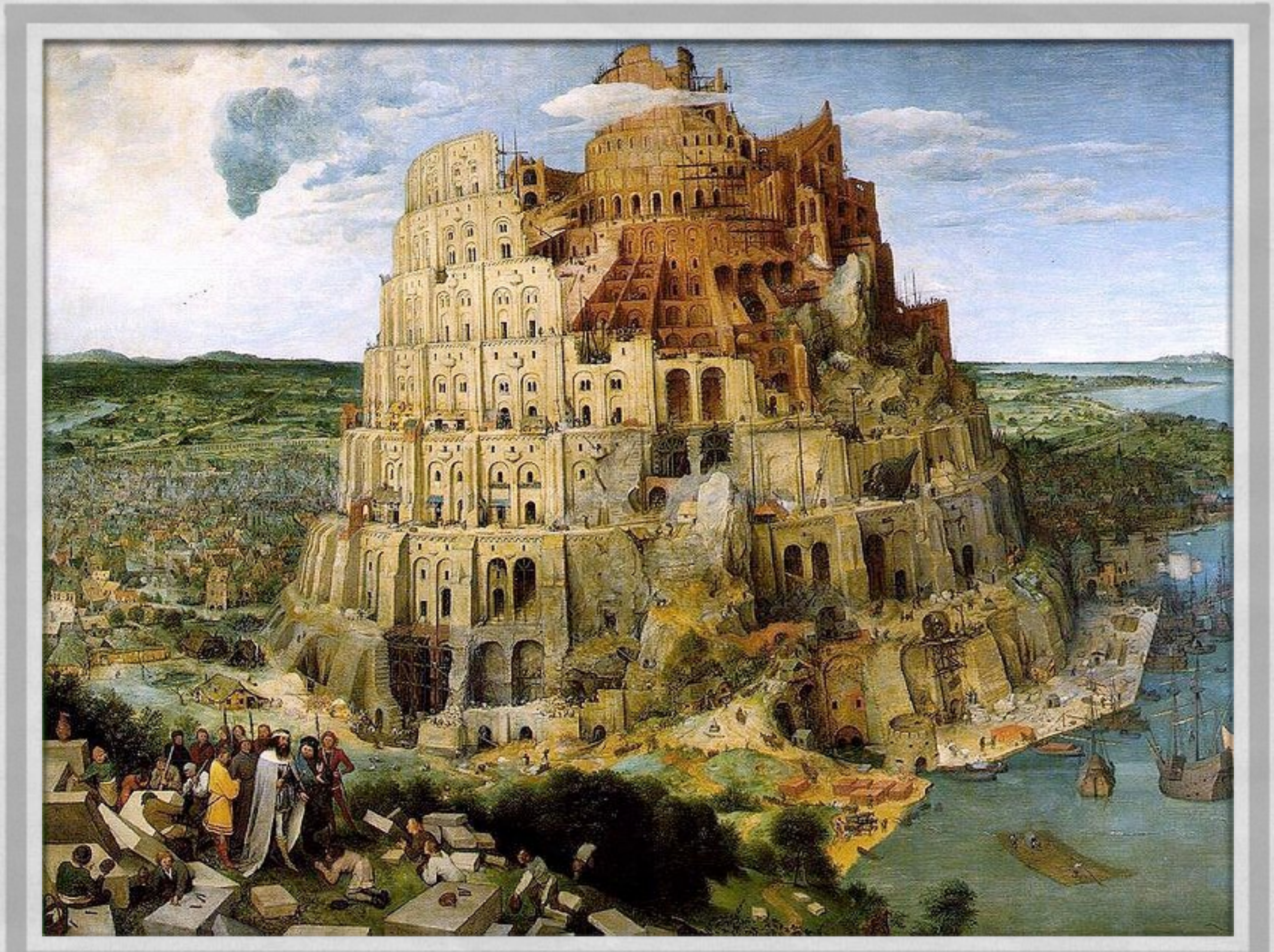
EMBEDDED SYSTEMS PROGRAMMING 2014-15

Language Basics

(PROGRAMMING) LANGUAGES



"The tower of Babel" by Pieter Bruegel the Elder
Kunsthistorisches Museum, Vienna

ABOUT THE LANGUAGES

- **C** (1972)
  - Designed "to replace assembly language" and still being efficient
  - Standard: ISO/IEC 9899:2011 (latest version, december 2011)

- **C++** (1983)
  - Designed to add object orientation to C while still allowing low-level (sometimes nasty) operations. 99.9% compatible with C.
  - Standard: ISO/IEC 14882:2011 (latest version, september 2011)

- **Java** (1993)
  - Designed to be easier and less error-inducing than C++
  - Standard: none, interested parties decide the way to follow via the JCP

PARADIGMS

The aforementioned languages can be considered

- **imperative**
  The program is composed by a series of statements that dictate what should be done
- **structured**
  Control structures (loops, etc.) are available
- **procedural**
  Control structures called "subroutines" are available
- for C++ and Java: **object-oriented**

OBJECT ORIENTATION

- Several modern programming languages embrace the **object-oriented** (OO) paradigm

- Data and code must/can be encapsulated into special structures called **objects**
  - Encourages associations with real-world entities, which should make programming easier
  - Favors code modularity
- More about OO programming in a few lessons

(C)
C++
JAVA

FORMATTING

- The following rules apply to all 3 languages (C, C++, Java)
- White spaces separate names and keywords
- Statements are terminated by a “ ; ”

COMMENTS

- The following rules apply to all 3 languages (C, C++, Java)

- Anything from "//" to the end of a line is a comment

- Anything enclosed between "/*" and "*/" is a comment

COMMENTS: JAVA

- In Java, a comment starting with **two asterisks** is a **documentation comment**

```
/** Sample documentation comment */
```

- A documentation comment describes the declaration that follows it
- Many IDEs are able to handle and/or extract documentation comments

NAMES

- The following rules apply to all 3 languages (**C**, **C++**, **Java**)
- A name includes letters, numbers and "_". The first character must be a letter
- No white spaces allowed inside a name
- Names are case sensitive

VARIABLES

- The following rules apply to all 3 languages **(C, C++, Java)**
- The languages are *statically-typed*: all variables must be *declared* before use
- A declaration contains the data type and the name of the variable
- A default value may be optionally specified

VARIABLES: INITIALIZATION

- **Java**: if no value is provided, variables are initialized to zero by default
- **C, C++**: if no value is provided, variables assume a random value

PRIMITIVE DATA TYPES (1/2)

- The following data types are common to all 3 languages (C, C++, Java)

- `short`: 16-bit signed two's complement integer
- `int`: 32-bit signed two's complement integer
- `float`: 32-bit IEEE 754 floating point
- `double`: 64-bit IEEE 754 floating point

} C, C++: 32-bit computer

PRIMITIVE DATA TYPES (2/2)

- The following data types are common to all 3 languages (**C**, **C++**, **Java**)
- **Enumerated type** (`enum`): a set of named values. Use `enum` types to represent a fixed set of constants known at compile time

PRIMITIVE DATA TYPES: JAVA

- **`byte`**: 8-bit signed two's complement integer
- **`boolean`**: only two values, i.e. `true` and `false`
- **`char`**: 16-bit Unicode character

- All the integer types are always signed

PRIMITIVE DATA TYPES: C, C++

- **`bool`**: only two values, i.e. `true` and `false`
- **`char`**: 8-bit character
- **`void`**: generic identifier, does not imply type

- Integer data types can be **`unsigned`**
- **Pointers** to data (more on this later)

PRIMITIVE DATA TYPES: EXAMPLES (1/2)

- All 3 languages:

```
short n = 0x1234;
int i = -100000;
double pi = 3.14;
enum g = {alpha, beta, gamma};
```

- Java:

```
boolean result = true;
char capitalC = 'C';
```

PRIMITIVE DATA TYPES: EXAMPLES (2/2)

- C and C++:

```
bool result = true;

unsigned short j = 60000;

int * p; // pointer to integer
```

ARRAYS

- The following rules apply to all 3 languages **(C, C++, Java)**
- An **array** is a container that holds a fixed number L of values of the same data type
- L is established when the array is created
- The `i`-th element of an array `A` is identified by **`A[i]`**, with `i` ranging from 0 (zero) to L-1

ARRAYS: EXAMPLES

- Definition of an array of integers in Java:

```
int[] A = new int[10];
int[] B = {3,4,7,6,2}; // L=5
```

- Definition of an array of integers in C and C++:

```
int A[10];
int B[]  = {3,4,7,6,2}; // L=5
```

STRINGS

- **Java**: Unicode character strings are a primitive data type handled through the **`String`** class.
  Once created, a `String` object cannot be changed.

- **C++**: no strings, but the standard **`string`** class emulates them via null-terminated arrays of `char`

- **C**: no strings, no libraries,
  only null-terminated arrays of `char`

STRINGS: EXAMPLES

- Java

```
String Greetings = "Hello";
```

- C++

```
string Greetings = "Hello";
string Greetings("Hello"); /* as above */
```

- C

```
char Greetings[] = "Hello"; /* 6 bytes */
```

CONSTANTS

- To declare a variable as constant
  - Java: prepend the `final` keyword
  - C, C++: prepend the `const` keyword

OPERATORS

- Common to all 3 languages (C, C++, Java)

- Assignment: `=`
- Arithmetic: `+` `-` `*` `/` `%` `++` `--`
- Bitwise: `&` `|` `~` `^` `<<` `>>`
- Relational: `==` `!=` `<=` `>=` `<` `>`
- Conditional: `&&` `||`

OPERATORS: JAVA

- The + operator is a **concatenation operator** when at least one of its operands is a string
(more about strings later)

OPERATORS: EXAMPLES

- The following expressions are equivalent

```
i = i + 1;

i++;

++i;

i += 1;
```

FUNCTIONS

- **Function**: piece of code that can be invoked to perform a specific task
- Identified by a function name
- Can receive one or more input parameters
- Can return at most one output parameter
- Java: no functions, only methods (e.g., functions inside a class)

DECLARATION VS. DEFINITION

- **Declaration**: only the name and parameters (i.e., the function **prototype**) are specified
- **Definition**: code for the function (i.e., the function **implementation**) is provided
- Declaration and definition can be provided together or kept separate
- *Mutatis mutandis*, the same can be said also for variables, methods, classes...

FUNCTIONS: EXAMPLES

- Declarations in C and C++

```
void f(void);

float generate_random_number(void);

void close_file(int file_id);


int sum(int a, int b);

int sum(short a, short b); // Functions with the same name can
                           // coexist as long as they have a
                           // different prototype ("overloading")
```

RETURN

- **C, C++, Java:**
  used to specify the return value of a function or a method
- Terminates the execution of the function/method

HEADER FILES (1/2)

- **C, C++**: contain declaration of variables and classes, prototypes of library functions, ...
  Use the `.h` extensions.
  Can be included (and therefore shared) by many source files.

- `#include` directive

EXAMPLE: C++

- **sum.h**: contains the declaration of function `sum`

```
#ifndef SUM_H             // To avoid multiple declarations
#define SUM_H
int sum(int a, int b);
#endif
```

- **sum.cpp**: contains the definition of function `sum`

```
#include "sum.h"

int sum(int a, int b)
{
    return a+b;
}
```

- **program.cpp**: uses function `sum`

```
#include "sum.h"

...

result = sum(quantity1, quantity2);

...
```

HEADER FILES (2/2)

- **Java**: no header files. Identifiers are automatically
  - extracted from source files,
  - read from dynamic libraries

PACKAGES AND NAMESPACES

- Java: **Package.** C++: **Namespace**
- Purpose: grouping names into contexts so as to avoid *naming collisions*
- You must use the *fully qualified name* of an element in a package/namespace, unless you previously declared that the package/namespace is being used

EXAMPLE: JAVA

```
package foo;

public class Global
{
    public static int bar;      // more on static later
}
```

In another source file:

```
import foo;             // import the package


++foo.Global.bar;       // fully qualified name

++Global.bar;           // short name
```

- Code not explicitly declared within a package goes into the *unnamed package*

EXAMPLE: C++

```
namespace foo
{
    int bar;       // Inside a namespace, but not inside a class
}
```

In another source file:

```
using namespace foo;  // import the namespace

++foo::bar;           // fully qualified name

++bar;                // short name
```

- Code not explicitly declared within a namespace goes into the *global namespace*

ENTRY POINT OF A PROGRAM

- Java: "`main(...)`" method of the entry class (can be specified if the program is inside a JAR)

- C, C++: "`main(...)`" function

- The "..." in "`main(...)`" indicates the program's parameters
- Syntax for parameters is fixed

"HELLO WORLD!": JAVA

- Hello.java

```
class Hello
{

    public static void main(String[] args)
    {
        System.out.printf("Hello World!\n");
    }

}
```

"HELLO WORLD!": C

- Hello.c

```
#include <stdio.h>

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

    printf("Hello World!\n");

    return 0;
}
```

"HELLO WORLD!": C++

- Hello.cpp

```
#include <stdio.h>

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

    printf("Hello World!\n");

    return 0;
}
```

“HELLO WORLD!”: TRUE C++

- Hello2.cpp

```
#include <iostream>

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

    std::cout << "Hello World!" << std::endl;

    return 0;

}
```

CONDITIONAL EXECUTION

- Common to all three languages

- **`if(…) {…} else {…}`** construct:
  the boolean condition inside `(…)` is calculated;
  if it evaluates to true, then the code inside the former pair of curly braces is executed, otherwise the code inside the latter pair

- The `else {}` part is optional: if it is not specified and the condition evaluates to false, no code is executed

EVALUATION RULE

- Beware of the evaluation rule for subclauses!

```
if( (c<10) || ((a==1) && (a<c++)) )  {…}
```

- **Short-circuit evaluation**: subclauses are evaluated from left to right and the evaluation stops as soon as the boolean value of the whole clause is univocally determined
- Can be an issue if some subclauses perform assignments or have other side effects

SWITCH(...)...CASE

- Common to all three languages

- The (non-boolean) expression following **`switch`** is evaluated, then the **`case`** clause associated with the value is executed

- No `case` for the value: no code is executed

- **`default`** keyword (optional): used to label a block of statements to be executed if no `case` matches

SWITCH(...)...CASE: EXAMPLE

```
switch(n)
{
    case 0:
        /* Code to execute when n is zero */
        break;

    case 1:
    case 4:
    case 9:
        /* Code to execute when n is a perfect square */
        break;

    case 3:
    case 5:
    case 7:
        /* Code to execute when n is a small prime number */
        break;

  default:
        /* Code to execute in all the remaining cases,
           for instance, when n=2 or n=8 or... */
        break;
}
```

Adapted from Wikipedia

LOOPS (1/3)

- Common to all three languages
- **`for(...)`** loop

```
for(var_init; exit_condition; var_incr)
{
    //code
}
```

- The loop is executed as long as the condition is true (possibly forever)

LOOPS (2/3)

- Common to all three languages
- **`while(...)`** loop

```
while(exit_condition)
{
     //code
}
```

- The loop is executed as long as the condition is true (possibly forever, possibly zero times)

LOOPS (3/3)

- Common to all three languages

- **`do...while(...)`** loop

```
do
{
      //code
}
while(exit_condition);
```

- The loop is executed as long as the condition is true (possibly forever, at least one time)

LOOPS: EXAMPLES

- C, C++, Java

```
for(i=0; i<10; i++) { A[i]=10-i; }

i = 0;
while(i<10) { B[i]=10-i; i++; }

i = 0;
do { C[i]=10-i; i++; } while(i<10);
```

- At the end of the program, A=B=C

BREAK

- Common to all three languages

- Terminates the execution of one of the following:
  - `switch(…)...case`
  - `for(…)` loop
  - `while(…)` loop
  - `do…while(…)` loop

BREAK: EXAMPLE

- A fourth way to initialize an array

```
i = 0;
while(1!=0)
{
    D[i]=10-i;
    i++;

    if(i >= 10) break;
}
```

GOTO

- **C and C++**: transfers execution to a specific source position, identified by a label

```
while(1)
{
    /* Do something */

    if(condition) goto foo;

    /* Do something else */
}

foo:
++v; // First line executed after the goto
```

- `goto` gained a bad name; it is seldom used nowadays
- **Java**: although reserved as a keyword, `goto` is not used and has no function

GOTO CONSIDERED HARMFUL

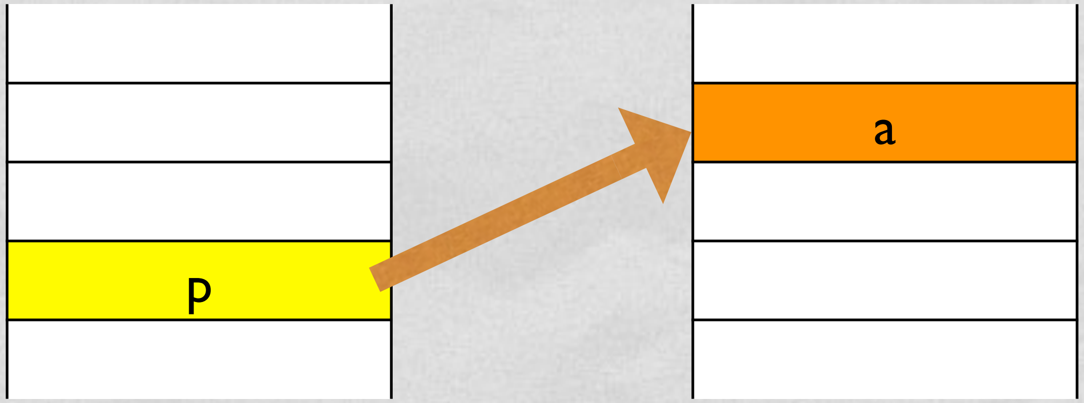
“For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of ‘go to’ statements they produce [...] The ‘go to’ statement should be abolished from all higher level programming languages”

Edsger W. Dijkstra
<u>*Communications of the ACM*</u>
March 1968

POINTERS (1/3)

- C and **C++** only. No pointers in Java!

- A pointer is a data type that do not contain data: it contains the address of data stored elsewhere



p is a pointer to a

POINTERS (2/3)

- Definition of a pointer

```
int * p;
```

- Assignment of an address to a pointer via the **reference operator** **&**

```
p = &a;
```

- Access to pointed data via the **dereference operator** *

```
b = *p;    // b=a
*p = 10;   // a=10
```

POINTERS (3/3)

- The size of a pointer is equal to the size of addresses on the host machine (nowadays, 32 or 64 bits)

- A pointer may be `NULL`
(i.e., it does not point to anything valid)

- If a pointer is not `NULL`, there is no way to tell whether it points to valid data or not

VOID POINTERS (1/2)

- **void pointers** point to a value that has **no type** (and thus also no specified length)
- void pointers can point to any kind of data but cannot be directly dereferenced

```
void f(void* data, int data_type)
{
    char * pc;
    int  * pi;
    if(data_type == 1){
        pc =(char*)data;  // cast to char
        // use data as char
    }
    else if(data_type == 2){
        pi = (int*)data;  // cast to int
        // use data as int
    }
}
```

VOID POINTERS (2/2)

- C allows implicit conversion from `void*` to other pointer types
- C++ does not
  (an example of incompatibility between C and C++)

```
void f(void* data, int data_type)
{
    char * pc;
    int  * pi;
    if(data_type == 1){
        pc = data;  // OK in C, not OK in C++
        // use data as char
    }
    else if(data_type == 2){
        pi = data;  // OK in C, not OK in C++
        // use data as int
    }
}
```

POINTER ARITHMETIC

- **C** and **C++** only

- Arithmetic operators can be applied to pointers

- When calculating a pointer arithmetic expression, the integer operands are multiplied by the size of the object being pointed to

```
int * p;
int * q = p-1;  // if sizeof(int)=4, q=p-4
p++;            // p=p+4
```

MALLOC, FREE

- C: *dynamic memory* must be allocated with the **`malloc`** stdlib function, and must be explicitly released with **`free`**
- C++: dynamic memory can be managed with the library functions malloc and free, or with the **`new`** and **`delete`** language operators

```
#include <stdlib.h>

...

unsigned char *color;     // A color in RGB format

color = (unsigned char *)malloc(3);
color[0] = color[1] = color[2] = 0;

...

free(color);
```