2.3.4.7 Data Memory Taxonomy

Return to Data Taxonomy

Overview

Return to Top

Data is tied to the memory where it is used during the Data-In-Use state. Basically, there are six different taxons that cover Data-In-Use. Although memory categories presented in Figure 1 are shown as having definite boundaries with definite rules governing the use of the categories; there are situations where these classification and boundaries deviate from the definiton. For example, the use dynamic code written as strings and processed at runtime, intepretive code, and reflection.

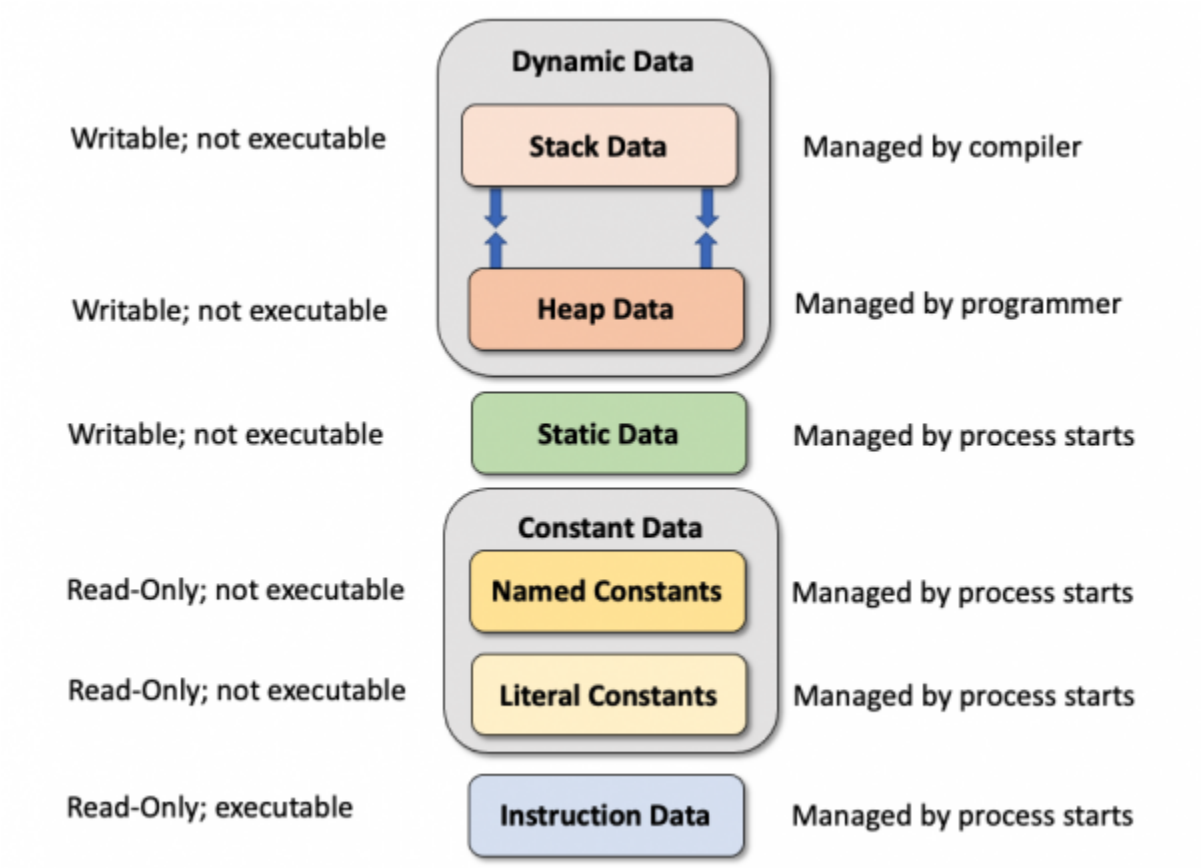

Figure 1: Traditional Memory Taxonomy

Instruction Data

Return to Top

A computer program is a ordered sequence of computer instructions that when executed in order accomplish a goal or task. Each computer architecture defines its own Instruction Set. When a program is executed, a Program Counter is set to the first instruction in the program. As the program executes each individual instruction, the program can modify data within the Variable area or it can modify the program counter (i.e., GO TO a specific instruction in the instruction set). As a general rule, the Instruction data can not be changed during the execution of a program. When the programs do change the contents of the excuting program, it is usually considered a fault and the program stops executing (see Segmentation Fault (SEGFAULT)).

Constant Data

Return to Top

Constant Data is a value that is expected to remain constant overtime (i.e., the number of pennies in a dollar, the number of seconds in a minute, the value of Pi, the gravitation constant, speed of light, etc.). In software, constants are usually hardcoded into the software and require a recompile and link in order to change them. Therefore, although in programming constants can change, it is generally not a trival task. Sometimes constants might need to change or updated is when the precision used to represent the constant value needs to change (i.e., the value of Pi going from single precison `3.1415927` to double precision `3.14159265358979323846`. Constants in programming should not change during the normal execution of the program. Sometimes the “constancy” canbe violated or pruposely circumvented by by overwriting the memory location where the value is stored or by using self-modifying code or dynamic code. Using constants in software can:

- Alow for compile-time or editing-time checking of operations such as assigning a value to a constant. In the following examle, most modern editors and all compilers raise erros on the second line.

```
const float PI = 3.1415927;
PI = 42;
```

- Increase reability (i.e., myVar*PI where PI is the value of the numerical constant 3.1415927 is more readable than myVar*3.1415927)

```
const float PI = 3.1415927;
float radius = 4;
float areaVar = 0;
areaVar = PI * (radius*radius); -- PI*r^2
```

- Icrease performance by aiding in compiler optimation (i.e., allocating memory once to hold the constant rather than once per usage)

```
const float PI = 3.1415927;
float radius1 = 4;
float radius2 = 6;
float areaVar1 = 0;
```

```
  float areaVar2 = 0;
  areaVar1 = 3.1415927 * (radius1*radius1);
  areaVar2 = 3.1415927 * (radius2*radius2);
```

There are two kinds of Constants used in most programming languages:

- Literal
- Named

Literal Constants

Return to Top

Literal Constansts are the actual numeric values (i.e., 3.1415927) or are characters (i.e., 'H'), or strings of characters (i.e., "Hello World") used in the software. In the following example, "Helo World! %d", 42 and 0 are literal constants.

```
int main()
{  cout<<"Hello World! %d", 42;
   return 0;
}
```

Named Constants

Return to Top

Named Constants are similar to variables in most programming languages, however, by convention many programmers will use-upper case names for the Constants to help the programmer visually "see" they are working with a Named Constant rather than a variable. In the following example, ANSWER and GREETING are named constants. "%s %d" and 0 are literal constants.

```
int main()
{  const int ANSWER = 42;
   const string GREETING = "Hello World";
   cout<<"%s %d", greeting, answerToUniverse;
   return 0;
}
```

Variable Data

Return to Top

Variable Data, in contrast to **Constant Data**, is a data item whose value can change during the

program's execution. In other words, the **Variable Data** value varies over time. There are two main categories based on the lifespan of the of **Variable Data**:

- Static Data
- Dynamic Data

Static Data

Return to Top

Static Data is data that does not change after it has been set. Static Data is usually defined during startup or initialization os a system or subsystem. Some comon examples of Static Data are setup parameters of a system; or properties or attributes of objects within a system (i.e., constructor settings). Data that is static differs form immutable data because immutable data can have newer values, but the original values are never replaced, lost or destroyed. Generally, newer immutable data has pointers to the privious values of the data.

It is a fixed data set. Experts contrast static data with dynamic data, where dynamic data may change after it is recorded, and has to be continually updated.

Dynamic Data

Return to Top

Dynamic Data is usually data is created, updated and maintained by the programs. It is generally divided into two areas, the **Heap** and the **Stack**. The two areas are organized and managed differently and the boundary between the two areas is fluid (i.e., the amount of space allocated to either the **Heap** or the **Stack** can expand and contract as long as the total amount of memory in the **Heap** or the **Stack** does not exceed the amount of **Dynamic Data** allocated for the program.

Stack Data

Return to Top

Stack Memory is generally allocated and unallocated from contiguous blocks of memory automatically during the execution of a program. When a program is executed, memory is allocated for all the parameters passed to the program and all the variables declared in the program. Upon the completion of the program, all the memory allocated for the program is released to be used by the next program. Often, the first program executed is often referred to as the **Main** program.

Any other programs (i.e., subprograms) called during the program are again allocated more memory for its parameters and declared variables from the **Stack**. Upon termination of the subprogram, the memory allocated to the subprogram is released back to the system to be reused for other subprograms. **Note:** the original memory allocated for the main (or calling program) program remains allocated until that

program ends. This repetitive and successive allocation creates a **stack** of allocation, with the most recent subprogram being at the top of the **stack**. The **stack** gets deeper and shallower depending on the depth of subprogram allocations.

Figure 2 illustrates a simple example of how the **stack** works:

1. A program is written into a file that contains two routines: **`greetings`** and a **`main`**
2. During compilation of the file, the compiler determines the amount of memory required to run **`main`** routine
3. During compilation of the file, the compiler determines the amount of memory required to run **`greetings`** routine
4. The **`main`** program is executed from the command lime and the amount of memory required to run the **`main`** is allocated and placed on the **stack**
5. When the **`main`** program executes the line that invokes the **`greetings`** routine, the memory required for it is allocated and place onto the **stack**
6. When the **`greetings`** routine is finished, the memory used to execute the routine is removed from the **stack** and released for re-use
7. When the **`main`** routine is finished, the memory used to execute the routine is removed from the **stack** and released for re-use

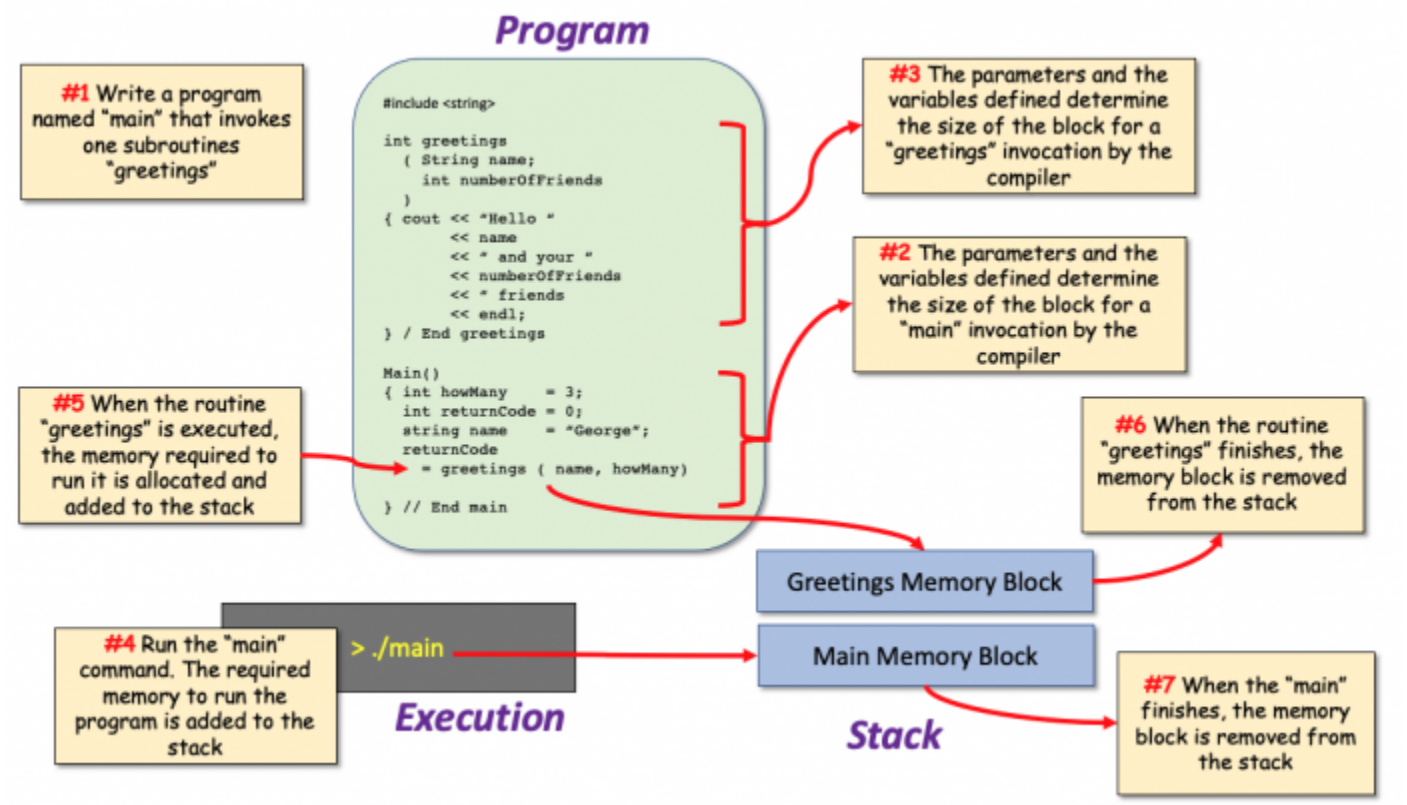

Figure 2: A Simple Example of **Stack** Memory

Heap Data

Return to Top

The Heap Memory is essentially a large pool of memory (typically per process) from which a running

program requests chunks of memory for use within the program. The memory is dynamically allocated during the execution of a program using allocation operations such as **`malloc`** or new operators on arrays or objects. When the program (i.e., process terminates, the chunks of memory allocated during the execution of the program are returned. Because the memory is allocated as it is needed and returned as it is required, the memory available for allocation often gets fragmented into small blocks which is very difficult to use and can result in memory exhaustion. This is why modern programming languages have implemented automatic memory management and a process called garbage collection.

1. A program is written into a file that contains a **`main`**. **Note:** During compilation of the file, the compiler determines the amount of memory required to run **`main`** routine
2. The **`main`** program is executed from the command lime and the amount of memory required to run the **`main`** is allocated and placed on the **stack**
3. When the **`main`** program executes the line that invokes the **`malloc`** statement, the memory required for it is allocated from the **heap**
4. When the **`main`** program executes the line that invokes the **`free`** statement, the memory required for it is returned to the *Heap **for later use (see - When the `main` program executes the line that invokes the next `malloc` statement, the memory required for it is allocated from the *Heap**. **Note:** This may or may not be the same memory allocated in the previous **`malloc`**.
5. When the **`main`** program executes the line that invokes the next **`free`** statement, the memory required for it is returned to the **Heap** for later use
6. When the **`main`** routine is finished, the memory used to execute the routine is removed from the **stack** and released for re-use

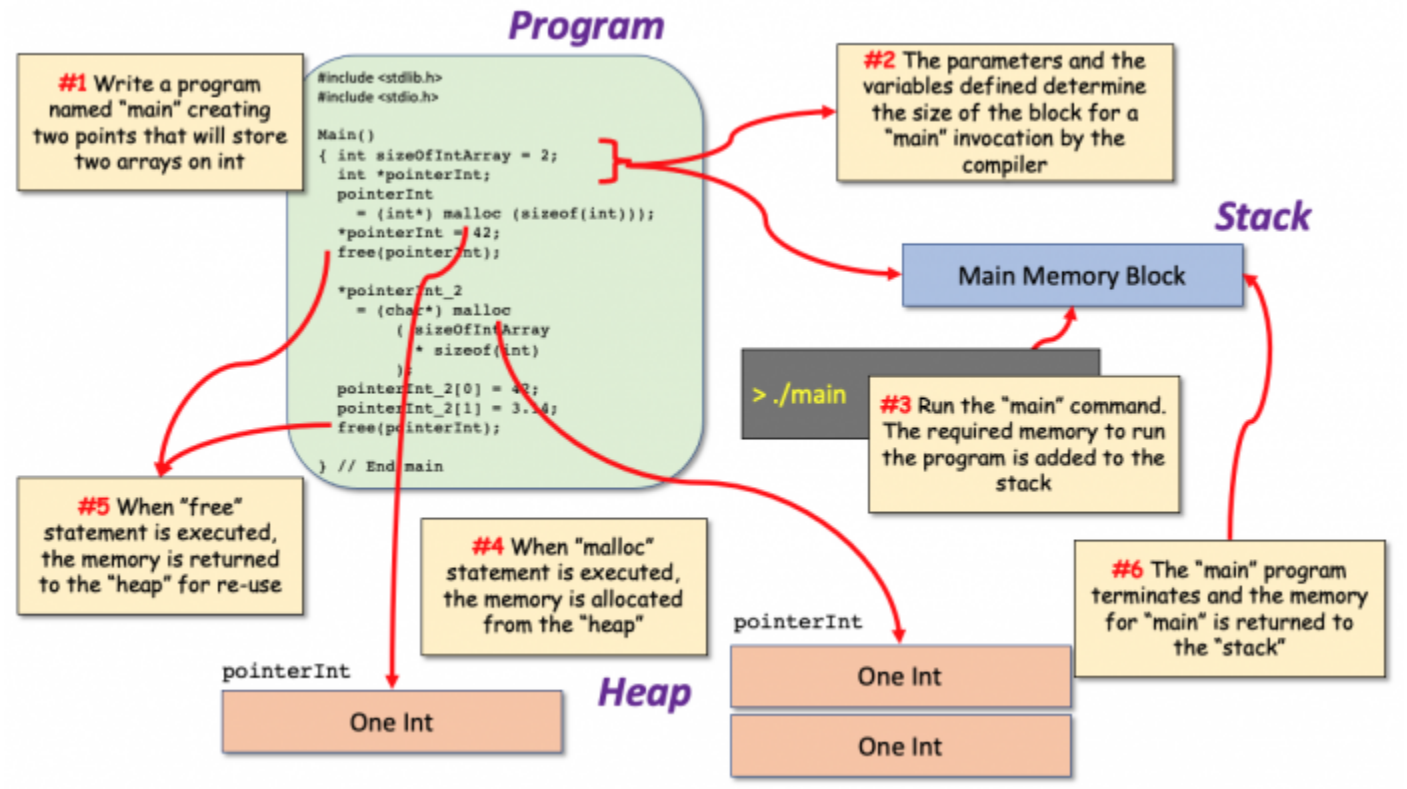

Figure 3: A Simple Example of **Heap** Memory

Attribute / Property Data

Return to Top

Often the two terms: **Attribute** and **Property** are used interchageable in Computer Science because we often use them to describe the elements of an Object in Object-Oriented Programming (OOP). However, it is useful to understand the difference between the two gramatically:

- An **Attribute** is a quality or object that we attribute to someone or something. For example, intelligence is an attribute of a person. You can not go to the store and buy intelligence, it is an attribute of the individual. Another example would be a planet belonging to a solar system. Planets do not usually exist outside a solar system.
- A **Property** is a quality that exists without any attribution. Geometric shapes are properties in their own right. They can be used to independently describe many things. For example, a planet or a ball can be described as a sphere[1)]. A definition of the sphere exists without the attribution to a planet or a ball. Therefore, we say that a planet or a ball have spherical properties.

In UML, an **Attribute** and a **Property** both represent a structural association between two entities. **Attributes** are most often represented as **Composition**[2)], while **Property** is best represented by **Aggregation** [3)].

In Unified Modeling Language (UML) according to Lenny Delligatti[4)], "UML Attributes are DataTypes (e.g., Strings, Integers, etc.) whereas Systems Modeling Language (SysML) properties are ValueTypes (e.g., can be assigned computed values)."

Some parameters are Mutable and others are Immutable. Mutable parameters can be changed inside an operation, and an immutable parameter can not. When a parameter is marked as **Immutable** (i.e., **`in`**) in some programming languages, a copy of the original value is made and passed to the operation on the Stack. One the operation is completed, the copied value is lost as the stack is unwound. Some languages such as the C, all values are passed by value (i.e., are **immutable**) unless the value is passed by reference using a pointer to the variable. When a parameter is marked as mutable, then a new values is returned. In C, the general way to do this is by returning the value via a **`return`** statement. If the **mutable** parameter is a pointer, then the value that the pointer points to can be changed, but not the pointer itself. In C, it is possible to have a ppointer to a pointer allowing the value pointed to be changed, but not the original pointer.

Table 1: Mutability of parameter values.

Parameter Diection	Internal Mutability	Description
in	immutable	An input Parameter (may not be modified).
out	mutable	An output Parameter (may be modified to communicate information to the caller).
inout	mutabe	An input Parameter that may be modified.
return	mutable	A return value of a call.

^ Fortran |

- Always used the inout semantics model
- Before Fortran 77: pass-by-reference
- Fortran 77 and later: scalar variables are often passed by value-result

|

C	• Pass-by-value • Pass-by-reference is achieved by using pointers as parameters
C++	• A special pointer type called reference type. Reference parameters are implicitly dereferenced in the function or method, and their semantics is pass-by-reference could have void fun(const int &p1, int p2, int &p3) { . . . } • p1 is pass-by-reference: p1 cannot be changed in the function fun • p2 is pass-by-value • p3 is pass-by-reference • Neither p1 nor p3 need be explicitly dereference in fun
Java	• All parameters are passed are passed by value • However, because objects can be accessed only through reference variables, object parameters are in effect passed by reference • Although an object reference passed as a parameter cannot itself be changed in the called subprogram, the referenced object can be changed if a method is available to cause the change
Ada	• Three semantics modes of parameter transmission: in, out, inout; in is the default mode • Formal parameters declared out can be assigned but not referenced; those declared in can be referenced but not assigned; inout parameters can be referenced and assigned
C#	• Default method: pass-by-value • Pass-by-reference can be specified by preceding both a formal parameter and its actual parameter with ref void sumer(ref int oldSum, int newOne) { . . . } . . . sumer(ref sum, newValue); ■ The first parameter to sumer is passed-by-reference; the second is passed-by-value
PHP:	very similar to C#
Perl:	all actual parameters are implicitly placed in a predefined array named @_

Python and Ruby	use pass-by-assignment (all data values are objects) • The process of changing the value of a variable with an assignment statement, as in x = x + 1 • does not change the object referenced by x. Rather, it takes the object referenced by x, increments it by 1, thereby creating a new object (with the value x + 1), and then x to reference the new object

☐ [nick]Is this page still under construction?
Page is still being updated – especially the sections above that are still blank
☐ [char]New Section -- review

1)
Sphere, In geometry, the set of all points in three-dimensional space lying the same distance (the radius) from a given point (the center), or the result of rotating a circle about one of its diameters. The components and properties of a sphere are analogous to those of a circle.https://www.britannica.com/science/sphere
2)
The **Composition** is a part of aggregation, and it portrays the whole-part relationship. It depicts dependency between a composite (parent) and its parts (children), which means that if the composite is discarded, so will its parts get deleted. It exists between similar objects.
https://www.javatpoint.com/uml-association-vs-aggregation-vs-composition
3)
Aggregation is a subset of association, is a collection of different things. It represents a relationship. It is more specific than an association and describes a part-whole or part-of relationship. Furthermore, it is a binary association, i.e., it only involves two classes. It is a kind of relationship in which the child is independent of its parent. https://www.javatpoint.com/uml-association-vs-aggregation-vs-composition
4)
Lenny Delligatti, SysML Distilled: A Brief Guide to the Systems Modeling Language, FIrst Edition, Addison-Wesley Professional , 8 November 2013, ISBN-13: 978-0321927866