# 6. Game Engine Support System (part. 3)

Game Engine Basics

Prof. H. Kang

# Keywords of prior knowledge

Hope these keywords help your study:

- Standard Template Library (STL)
- Booster Library
- C++ Template
- Cache Memory

# Keywords of prior knowledge

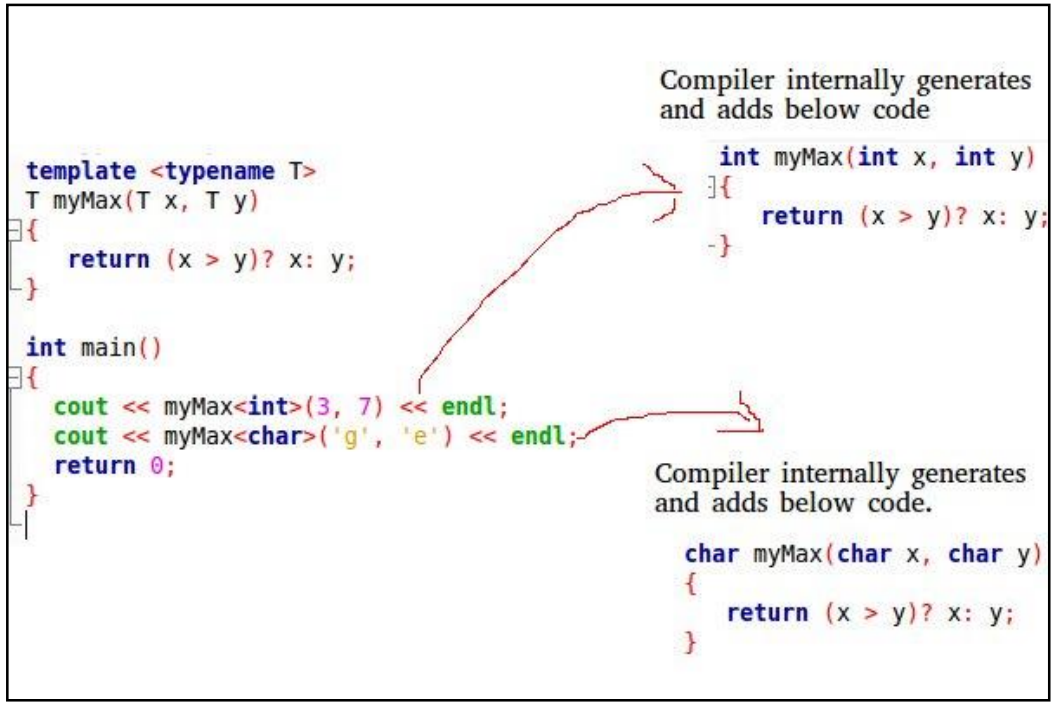
## Template:

- Function templates are special functions that can operate with generic types.
- This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

```
int class mypair {
 int values [2];
public:
  mypair (int first, int second) {
    values[0]=first;
    values[1]=second;
  }
};
```

```
double class mypair {
 double values [2];
public:
  mypair (double first, double second) {
    values[0]=first;
    values[1]=second;
  }
};
```

```
template <class T> class mypair {
  T values [2];
public:
  mypair (T first, T second) {
    values[0]=first;
    values[1]=second;
  }
};
```



# Introduction

## Implementation and use

- In previous lectures, we have learned many technique to handle many issues: construction & destruction, memory management, etc.

- To achieve them, we may need appropriate data structures, data types, and functions to handle data. In other words, appropriate **containers** are needed.
  - A container is a **holder object** that stores a collection of other objects (its elements). In C++, they are implemented as class templates, which allows a great flexibility in the types supported as elements.
  - The container manages the storage space for its elements and provides member functions to access them.

# Introduction

## Implementation and use

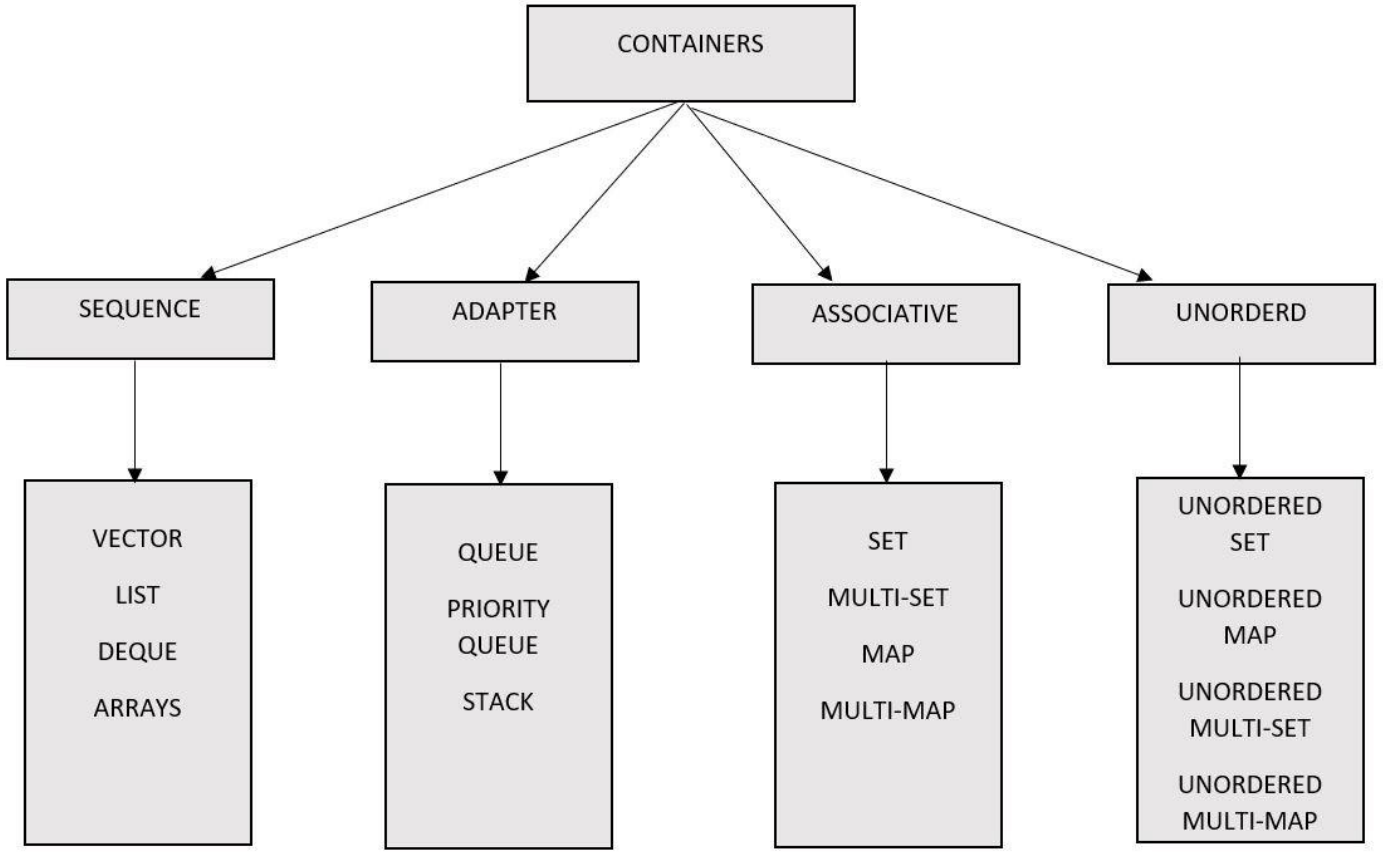
- Then, do we have to implement containers by ourselves?

- As game engine designers, we have number of choices:
  - Implement the needed data structures manually.
  - Rely on third-party implementations. Some common choices include
    a. the C++ standard template library (STL),
    b. a variant of STL, such as STLport,
    c. the powerful and robust Boost libraries.

# STL

## STL

- The benefits of the standard template library include:
  - STL offers a rich set of features.
  - Reasonably robust implementations are available on a wide variety of platforms.
  - STL comes standard with virtually all c++ compilers.



# STL

- The drawbacks are as follows:
  - STL has a steep learning curve. (The documentation is now quite good, but the header files are cryptic and difficult to understand on most platforms.)
  - STL is often slower than a data structure that has been crafted specifically for a particular problem.
  - STL also almost always eats up more memory than a custom-designed data structure.
  - STL does a lot of dynamic memory allocation, and it’s sometimes challenging to control its appetite for memory in a way that is suitable for high-performance, memory-limited console games.
  - STL’s implementation and behavior varies slightly from compiler to compiler, making its use in multiplatform engines more difficult.

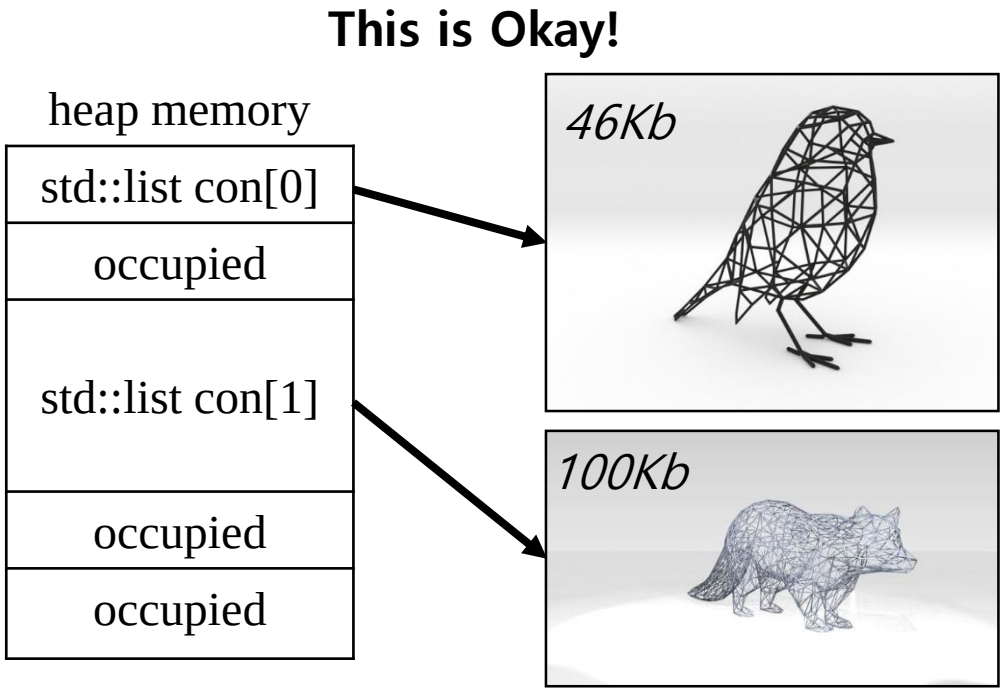
# STL

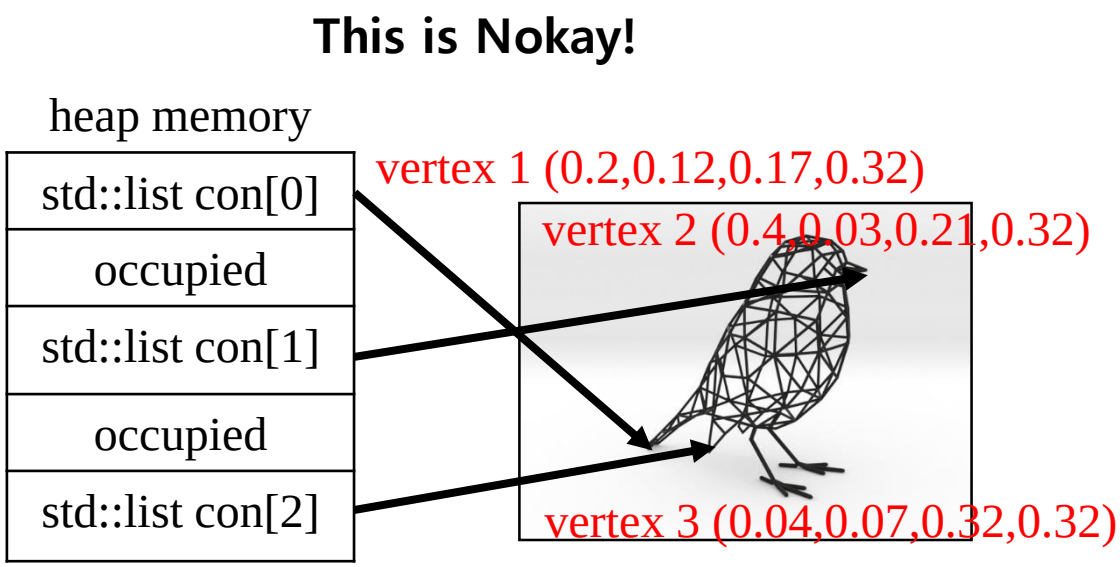
## STL

- STL is best suited to a game engine that will run on a personal computer platform, because the advanced virtual memory systems on modern PCs make memory allocation cheaper, and the probability of running out of physical RAM is often negligible.

# STL

- On the other hand, STL is not generally well-suited for use on memory limited consoles that lack advanced CPUs and virtual memory.
  - For example, embedding a *std::list* inside a game object is OK, but embedding a std::list inside every vertex of a 3D mesh is probably not a good idea. The *std::list* class dynamically allocates a small link object for every element inserted into it, and that can result in a lot of tiny, fragmented memory allocations.



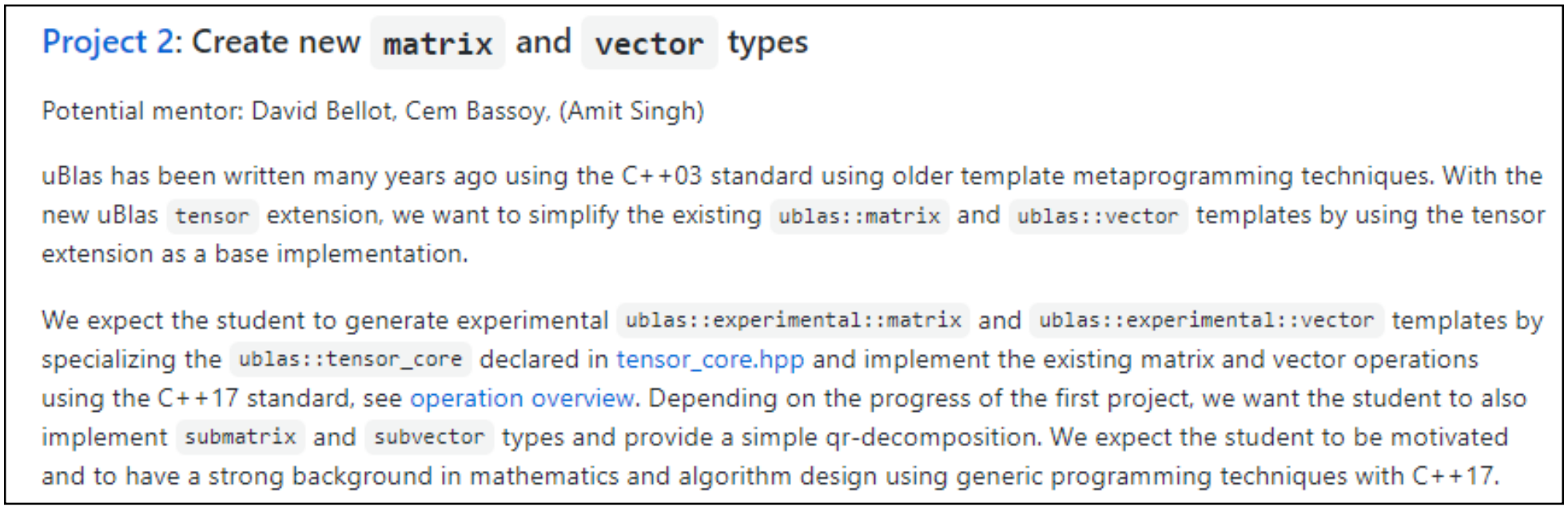
# Boost

## Boost

- Boost aims to produce libraries that extend and work together with STL, for both commercial and non-commercial use.
- Many of the Boost libraries have already been included in the C++ standards.
- If you are already using STL, then Boost can serve as an excellent extension and/or alternative to many of STL’s features.

# Boost

- The Boost libraries' documentation is usually excellent. Not only does the documentation explain what each library does and how to use it, but in most cases it also provides an excellent in-depth discussion of the design decisions, constraints and requirements that went into constructing the library. As such, reading the Boost documentation is a great way to learn about the principles of software design.



### Project 2: Create new `matrix` and `vector` types

Potential mentor: David Bellot, Cem Bassoy, (Amit Singh)

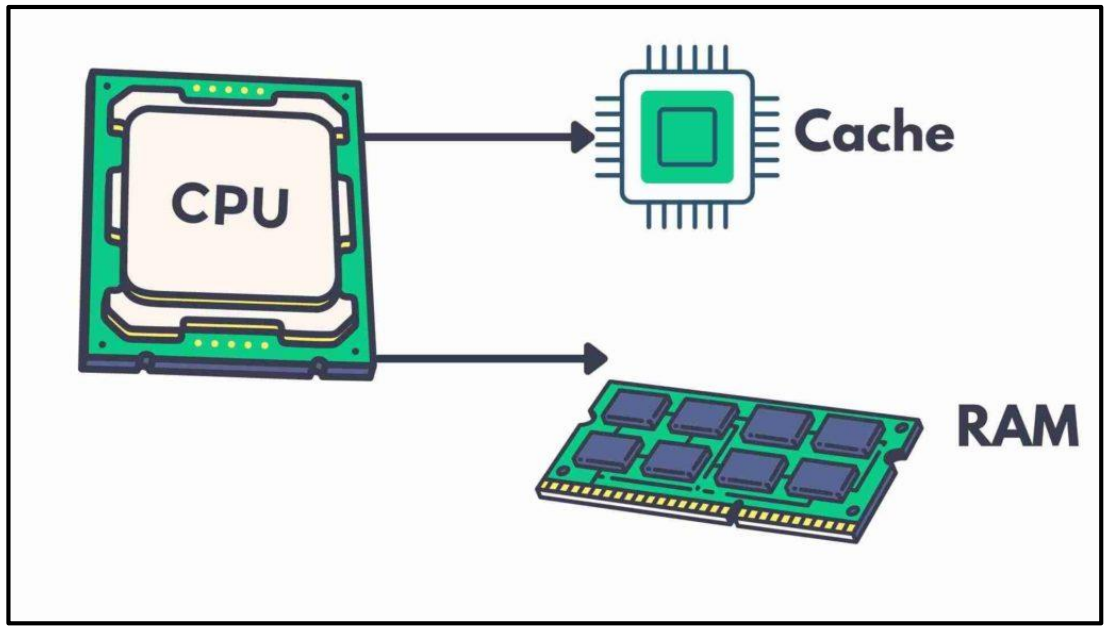
uBlas has been written many years ago using the C++03 standard using older template metaprogramming techniques. With the new uBlas `tensor` extension, we want to simplify the existing `ublas::matrix` and `ublas::vector` templates by using the tensor extension as a base implementation.

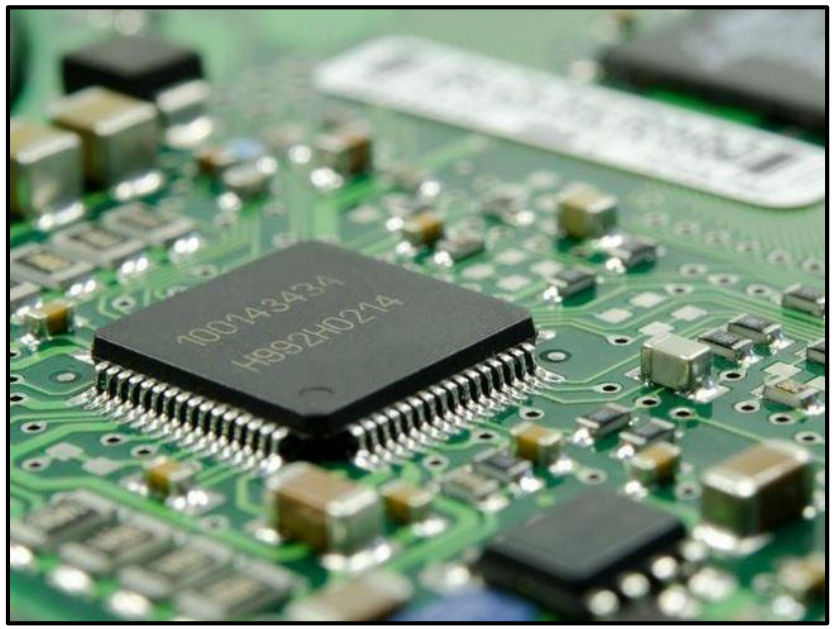
We expect the student to generate experimental `ublas::experimental::matrix` and `ublas::experimental::vector` templates by specializing the `ublas::tensor_core` declared in tensor_core.hpp and implement the existing matrix and vector operations using the C++17 standard, see operation overview. Depending on the progress of the first project, we want the student to also implement `submatrix` and `subvector` types and provide a simple qr-decomposition. We expect the student to be motivated and to have a strong background in mathematics and algorithm design using generic programming techniques with C++17.

# Containers

## Cache and fixed size array

- Fixed size C-style arrays are used quite a lot in game programming, because they require no memory allocation, are contiguous and hence cache-friendly, and support many common operations such as appending data and searching very efficiently.
- A CPU cache is a hardware cache used by the CPU of a computer to reduce the average cost to access data from the main memory.
- A cache is a smaller, faster memory, located closer to a processor core, which stores copies of the data from frequently used main memory locations.

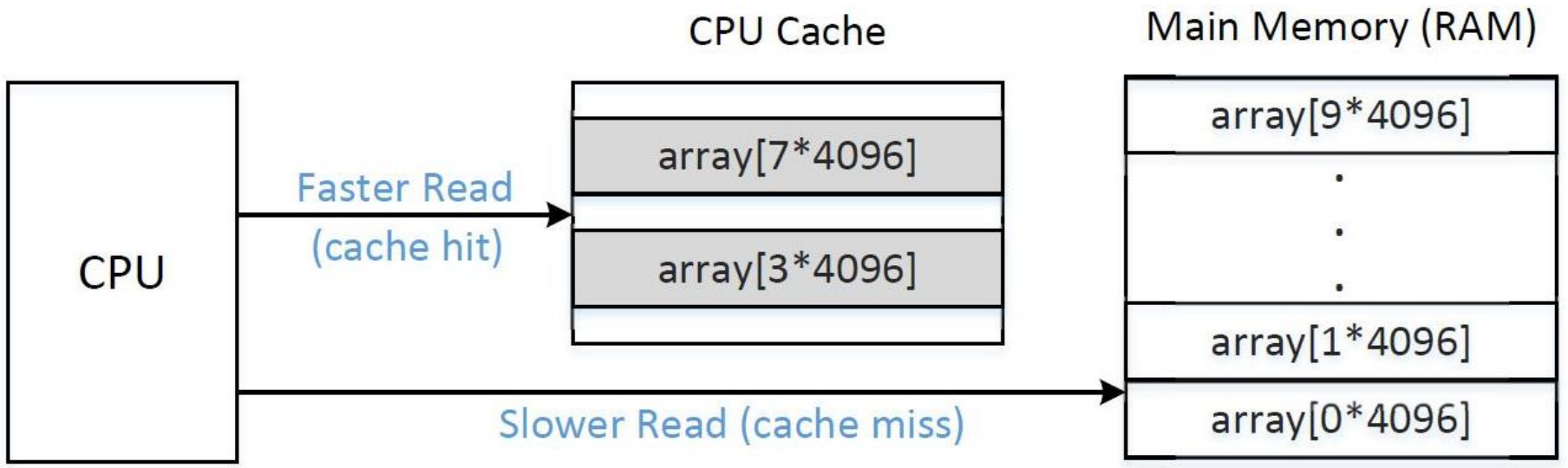


# Containers

## Cache and fixed size array

- If CPU needs *Arr[0]* to *Arr[7]* elements, the whole block that contains this element will be brought to cache. Since the access to the cache is surprisingly fast, overall performance increases.



# Containers

## Dynamic arrays and chunky allocation

- However, when the size of an array cannot be determined a priori, programmers tend to turn to *dynamic arrays*.
- The easiest way to implement a dynamic array is to allocate an *n*-element buffer initially and then *grow* the list only if an attempt is made to add more than *n* elements to it.
- This gives us the favorable characteristics of a fixed size array but with no upper bound.
- The STL *std::vector* class works in this manner.

### std::vector

Defined in header <vector>

```
template<
    class T,
    class Allocator = std::allocator<T>
> class vector;
```
(1)

```
namespace pmr {
    template <class T>
    using vector = std::vector<T, std::pmr::polymorphic_allocator<T>>;
}
```
(2) (since C++17)

# Containers

## Linked lists

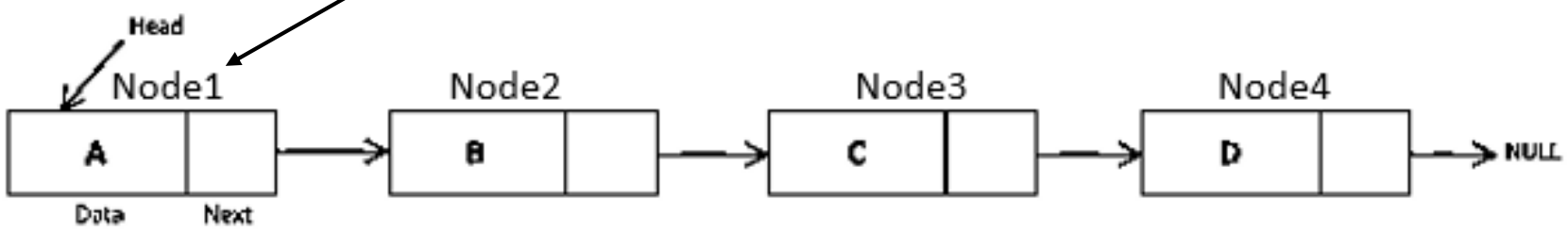
- If contiguous memory is not a primary concern, but the ability to insert and remove elements at random is paramount, then a linked list is usually the data structure of choice.
- Linked lists are quite easy to implement, but they’re also quite easy to get wrong.

# Containers

## The basics of linked lists

- Each element in the list has a pointer to the next element, and, in a doubly-linked list, it also has a pointer to the previous element.
- These two pointers are referred to as links. The list as a whole is tracked using a special pair of pointers called the *head* and *tail* pointers.
- The *head* pointer points to the first element, while the *tail* pointer points to the last element.

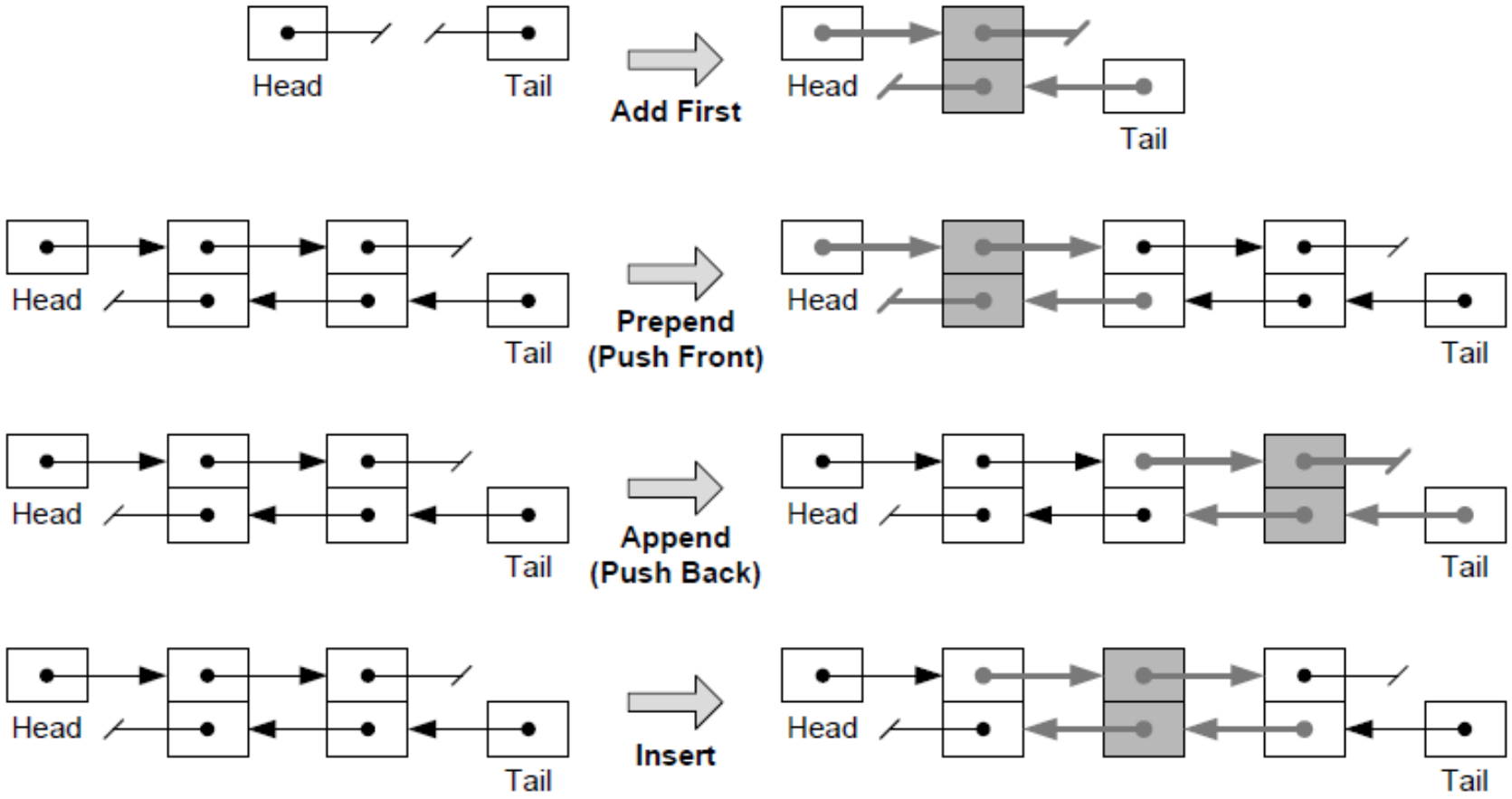
```
struct node
{
  int data;
  struct node *next;
};
```



# Containers

## The basics of linked lists

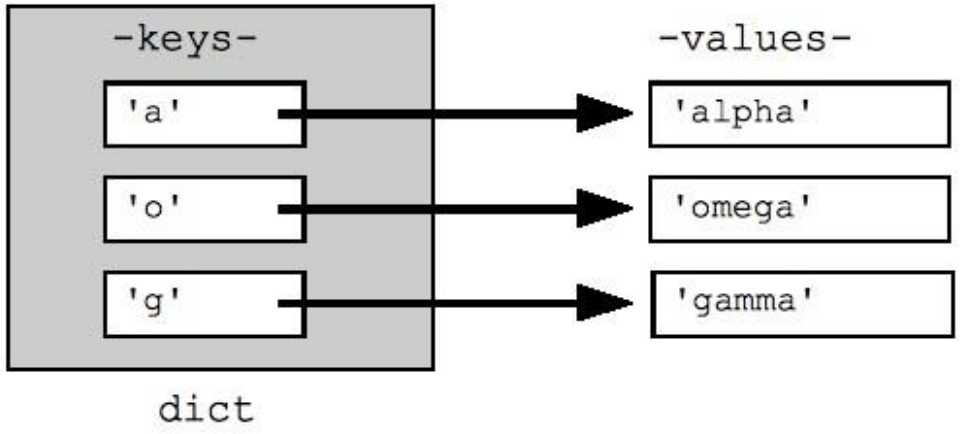
- Inserting a new element into a doubly-linked list involves adjusting the next pointer of the previous element and the previous pointer of the next element to both point at the new element and then setting the new element's next and previous pointers appropriately as well.

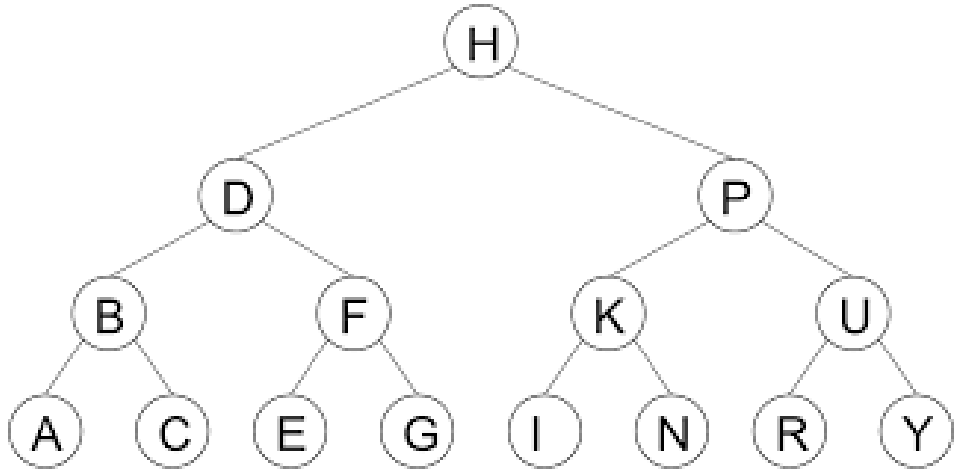


# Containers

## Dictionaries and hash tables

- A dictionary is a table of key-value pairs.
- A value in the dictionary can be looked up quickly, given its key.
- This kind of data structure can be implemented with a binary search tree.
  - The key-value pairs are stored in the nodes of the binary tree, and the tree is maintained in key-sorted order. Looking up a value by key involves performing an O(log *n*) binary search.

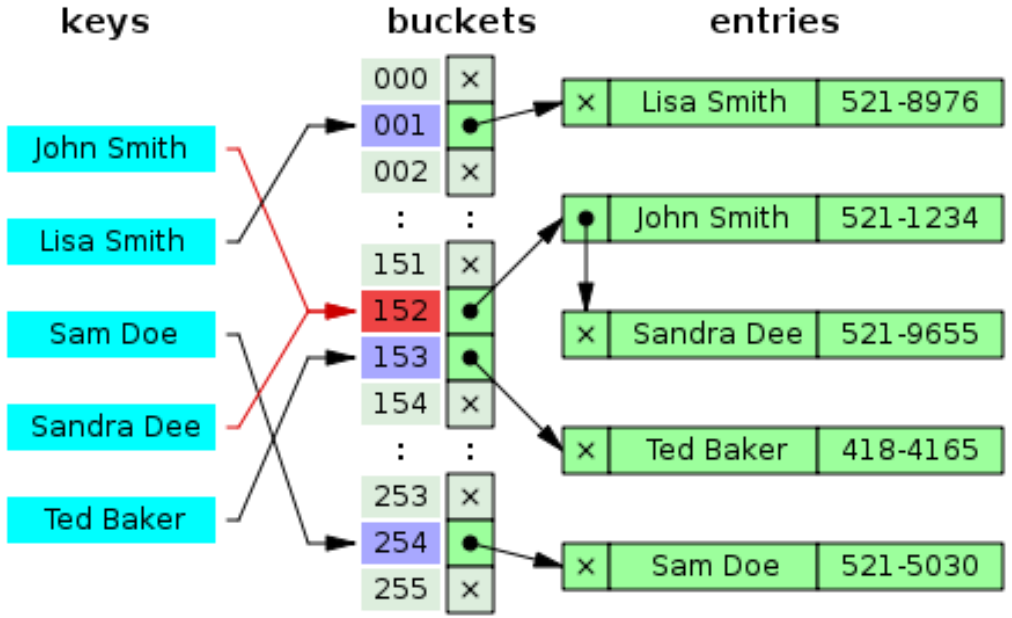


# Containers

Dictionaries and hash tables

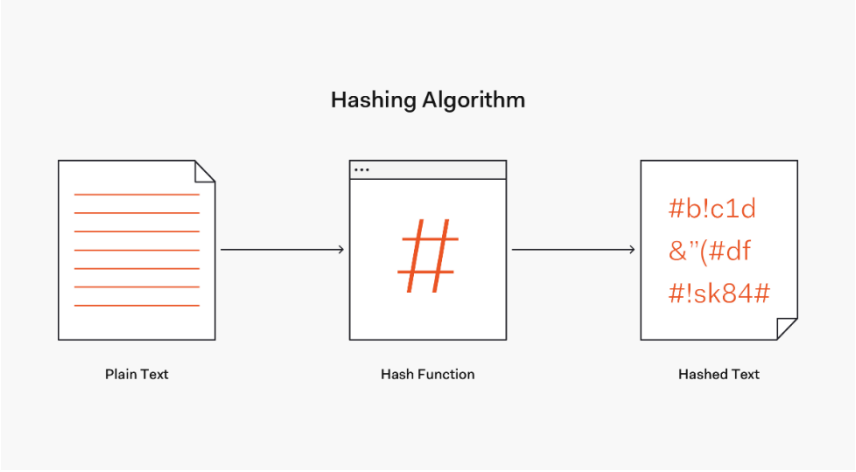
- This kind of data structure can be implemented with a hash table.
  - The values are stored in a fixed size table, where each slot in the table represents one or more keys.
  - To insert a key value pair into a hash table, the key is first converted into integer form via a process known as hashing (if it is not already an integer). Then an index into the hash table is calculated by taking the hashed key modulo the size of the table. Finally, the key-value pair is stored in the slot corresponding to that index.



# Containers

## Hashing

- Hashing is the process of turning a key of some arbitrary data type into an integer, which can be used modulo the table size as an index into the table.
- Mathematically, given a key *k*, we want to generate an integer hash value *h* using the hash function *H* and then find the index *i* into the table as follows:
  - $h = \mathrm{H}(k)$,
  - $i = h \bmod (N)$, where N is the number of slots in the table.
- If the key is a string, we can employ a string hashing function, which combines the ASCII or UTF codes of all the characters in the string into a single 32-bit integer value.
- The quality of the hashing function is crucial to the efficiency of the hash table.



# Containers

## Hashing

- A good hashing function is one that distributes the set of all valid keys evenly across the table, thereby minimizing the likelihood of collisions.
- Furthermore, a hash function must be reasonably quick to calculate, and deterministic in the sense that it must produce the exact same output every time it is called with an identical input.
- Here are a few functions:
  - LOOKUP3: http://burtleburtle.net/bob/c/lookup3.c
  - Cyclic redundancy check (CRC-32): https://en.wikipedia.org/wiki/Cyclic_redundancy_check
  - Message-digest algorithm 5 (MD5): https://en.wikipedia.org/wiki/MD5

# Strings

## Strings

- Strings are ubiquitous in almost every software project, and game engines are no exception.

## Problems with strings

- The most fundamental question about strings is how they should be stored and managed in your program.
- In C and C++, strings are not even an atomic type - they are implemented as arrays of characters.
  - Atomic? - Atomic types are those for which reading and writing are guaranteed to happen in a single instruction. In practice, *int* is atomic. There's no way for a handler to run in the middle of an access to *int*.
  - Comparing or Copying *int*s or *flost*s can be accomplished via simple machine language instructions.
  - On the other hand, comparing *string*s requires an *O*(n) scan of the character arrays using a function like *strcmp*().

# Strings

## Problems with strings (cont.)

- Another big string-related issue is localization - the process of adapting your software for release in other languages.
  - Any string that you display to the user in English must be translated into whatever languages you plan to support.
  - This not only involves making sure that you can represent all the character glyphs of all the languages you plan to support, but it also means ensuring that your game can handle different text orientations (Write horizontally, vertically, left-to-right, or right-to-left).

# Strings

## String classes

- String classes can make working with strings much more convenient for the programmer.
- However, a string class can have hidden costs that are difficult to see until the game is profiled.
  - For example, passing a string to a function using a C-style character array is fast because the address of the first character is typically passed in a hardware register.
  - On the other hand, passing a string object might incur the overhead of one or more copy constructors, if the function is not declared or used properly.
- For this reason, in game programming, programmers generally like to avoid string classes.
- If needed, make sure you pick or implement one that has acceptable runtime performance characteristics.

# Strings

## Unique identifiers

- Unique object identifiers allow game designers to keep track of the myriad objects that make up their game worlds and also permit those objects to be found and operated on at runtime by the engine.
- The assets from which our game objects are constructed all need unique identifiers as well.

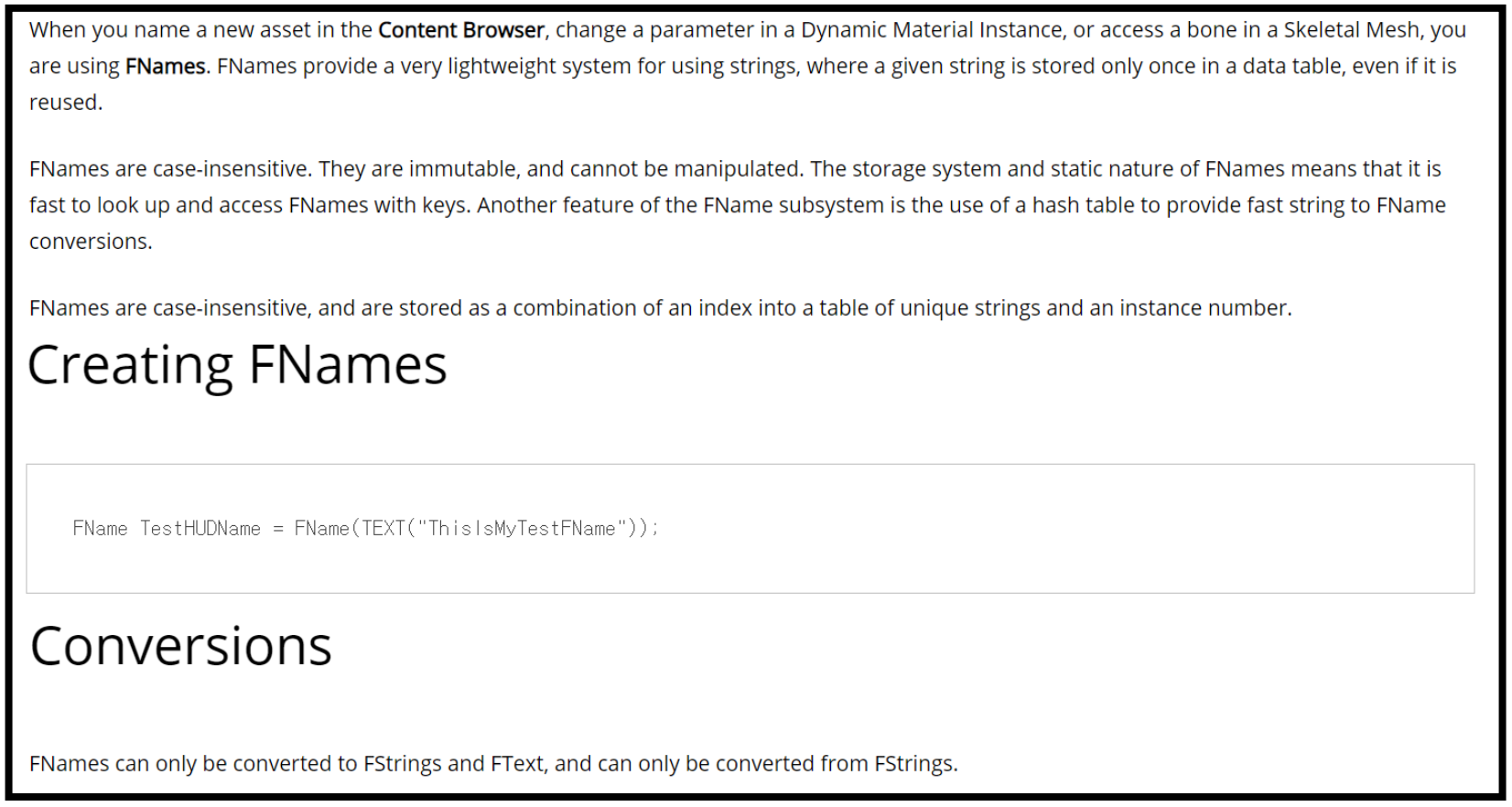
## Strings as unique identifiers

- Assets are often stored in individual files on disk, so they can usually be identified uniquely by their file paths, which of course are strings.
- Game objects are created by game designers, so it is natural for them to assign their objects understandable string names.
- However, the speed with which comparisons between unique identifiers can be made is of paramount importance in a game.
- We need a way to get all the descriptiveness and flexibility of a string, but with the speed of an integer.

# Strings

## Hashed string IDs

- A hash function maps a string onto a semi-unique integer.
- String hash codes can be compared just like any other integers, so comparisons are fast.
- Game programmers sometimes use the term *string id* to refer to such a hashed string. The Unreal engine uses the term *name* instead (implemented by class *FName*).



When you name a new asset in the **Content Browser**, change a parameter in a Dynamic Material Instance, or access a bone in a Skeletal Mesh, you are using **FNames**. FNames provide a very lightweight system for using strings, where a given string is stored only once in a data table, even if it is reused.

FNames are case-insensitive. They are immutable, and cannot be manipulated. The storage system and static nature of FNames means that it is fast to look up and access FNames with keys. Another feature of the FName subsystem is the use of a hash table to provide fast string to FName conversions.

FNames are case-insensitive, and are stored as a combination of an index into a table of unique strings and an instance number.

### Creating FNames

```
FName TestHUDName = FName(TEXT("ThisIsMyTestFName"));
```

### Conversions

FNames can only be converted to FStrings and FText, and can only be converted from FStrings.

# Strings

Localization and Unicode

- The problem of most software developers is that they are trained to think of strings as arrays of eight-bit ASCII character codes (i.e., characters following the ANSI standard).
- However, there are more complex languages that cannot be handled by ANSI standard.
- To address this limitations, the Unicode character set system was devised.
- The basic idea behind Unicode is to assign every character or glyph from every language in common use around the globe to a unique hexadecimal code known as a code point.

**ASCII/8859-1 Text**

| | |
|---|---|
| A | 0100 0001 |
| S | 0101 0011 |
| C | 0100 0011 |
| I | 0100 1001 |
| I | 0100 1001 |
| / | 0010 1111 |
| 8 | 0011 1000 |
| 8 | 0011 1000 |
| 5 | 0011 0101 |
| 9 | 0011 1001 |
| - | 0010 1101 |
| 1 | 0011 0001 |
| | 0010 0000 |
| t | 0111 0100 |
| e | 0110 0101 |
| x | 0111 1000 |
| t | 0111 0100 |

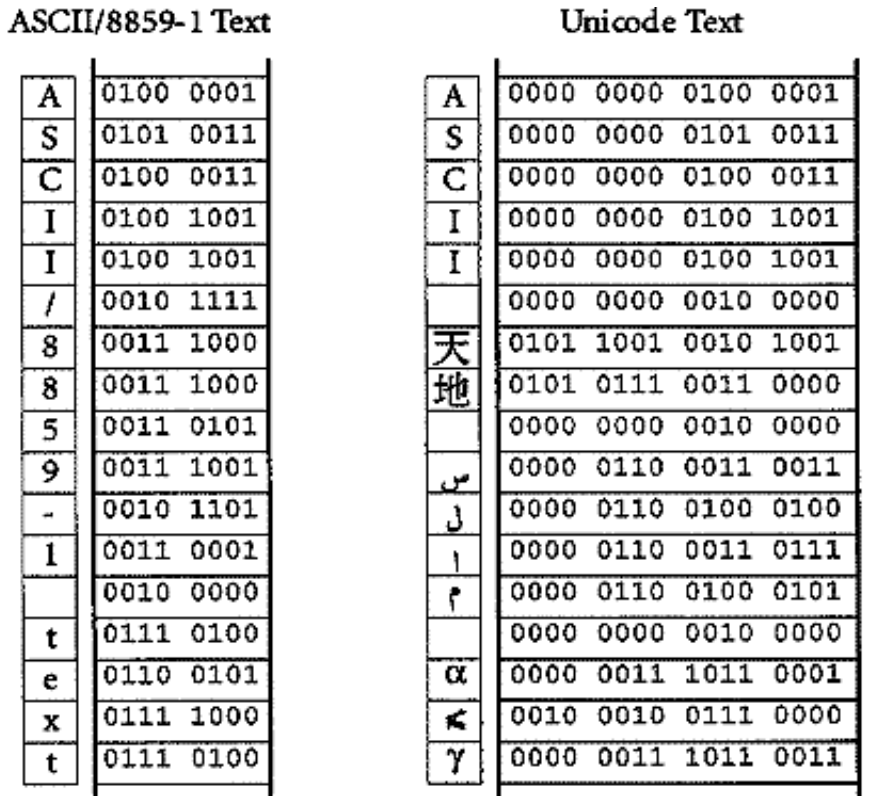
**Unicode Text**

| | |
|---|---|
| A | 0000 0000 0100 0001 |
| S | 0000 0000 0101 0011 |
| C | 0000 0000 0100 0011 |
| I | 0000 0000 0100 1001 |
| I | 0000 0000 0100 1001 |
| | 0000 0000 0010 0000 |
| 天 | 0101 1001 0010 1001 |
| 地 | 0101 0111 0011 0000 |
| | 0000 0000 0010 0000 |
| س | 0000 0110 0011 0011 |
| ل | 0000 0110 0100 0100 |
| ا | 0000 0110 0011 0111 |
| م | 0000 0110 0100 0101 |
| | 0000 0000 0010 0000 |
| α | 0000 0011 1011 0001 |
| ϰ | 0010 0010 0111 0000 |
| γ | 0000 0011 1011 0011 |

# Strings

## Localization and Unicode

- When storing a string of characters in memory, we select a particular encoding - a specific means of representing the Unicode code points for each character - and following those rules, we lay down a sequence of bits in memory that represent the string.
- UTF-8 and UTF-16 are two common encodings.



ASCII/8859-1 Text

| | |
|---|---|
| A | 0100 0001 |
| S | 0101 0011 |
| C | 0100 0011 |
| I | 0100 1001 |
| I | 0100 1001 |
| / | 0010 1111 |
| 8 | 0011 1000 |
| 8 | 0011 1000 |
| 5 | 0011 0101 |
| 9 | 0011 1001 |
| - | 0010 1101 |
| l | 0011 0001 |
| | 0010 0000 |
| t | 0111 0100 |
| e | 0110 0101 |
| x | 0111 1000 |
| t | 0111 0100 |

Unicode Text

| | |
|---|---|
| A | 0000 0000 0100 0001 |
| S | 0000 0000 0101 0011 |
| C | 0000 0000 0100 0011 |
| I | 0000 0000 0100 1001 |
| I | 0000 0000 0100 1001 |
| | 0000 0000 0010 0000 |
| 天 | 0101 1001 0010 1001 |
| 地 | 0101 0111 0011 0000 |
| | 0000 0000 0010 0000 |
| س | 0000 0110 0011 0011 |
| ل | 0000 0110 0100 0100 |
| ا | 0000 0110 0011 0111 |
| م | 0000 0110 0100 0101 |
| | 0000 0000 0010 0000 |
| α | 0000 0011 1011 0001 |
| ϰ | 0010 0010 0111 0000 |
| γ | 0000 0011 1011 0011 |

# Strings

## Unicode encodings (UTF-8, -16, -32)

- The simplest Unicode encoding is UTF-32.
  - Each Unicode code point is encoded into a 32-bit (4-byte) value.
  - This encoding wastes a lot of space:
    1) Most strings in Western European languages do not use any of the highest-valued code points, so an average of at least 16bits is usually wasted per character.
    2) The highest Unicode code point is 0x10FFFF, so even if we wanted to create a string that uses every possible Unicode glyph, we'd still only need 21 bits per character, not 32.

| character | encoding | bits |
|---|---|---|
| A | UTF-32 | 00000000 00000000 00000000 01000001 |
| あ | UTF-32 | 00000000 00000000 00110000 01000010 |

# Strings

## Unicode encodings (UTF-8, -16, -32)

- In UTF-8, the code points for each character in a string are stored using eight-bit.
  - However, some code points occupy more than one byte.
  - Hence the number of bytes occupied by a UTF-8 character string is not necessarily the length of the string in characters (This is known as a variable-length encoding, or a multibyte character set - MBCS).
  - One of the big benefits of the UTF-8 encoding is that it is backwards compatible with the ANSI encoding (This works because the first 127 Unicode code points correspond numerically to the old ANSI character codes. )

| character | encoding | bits |
|---|---|---|
| A | UTF-8 | 01000001 |
| あ | UTF-8 | 11100011 10000001 10000010 |

# Strings

## Unicode encodings (UTF-8, -16, -32)

- Each character in UTF-16 string is represented by either one or two 16-bit values.
  - In UTF-16, the set of all possible Unicode code points is divided into 17 planes containing $2^{16}$ code points each.
  - The first plane is known as the basic multilingual plane (BMP) which contains the most commonly used code points across a wide range of languages.
  - Many UTF-16 strings can be represented by code points within the first plane, meaning that each character in such a string is represented by only one 16-bit value.
  - However, if a character from one of the other planes is required, it is represented by two consecutive 16-bit values.

V·T·E **Unicode planes, and code point ranges used** [hide]

| Basic | | Supplementary | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Plane 0 | | Plane 1 | | Plane 2 | | Plane 3 | Planes 4–13 | Plane 14 | Planes 15–16 |
| 0000–FFFF | | 10000–1FFFF | | 20000–2FFFF | | 30000–3FFFF | 40000–DFFFF | E0000–EFFFF | F0000–10FFFF |
| Basic Multilingual Plane | | Supplementary Multilingual Plane | | Supplementary Ideographic Plane | | Tertiary Ideographic Plane (unassigned) | *unassigned* | Supplementary Special-purpose Plane | Supplementary Private Use Area planes |
| BMP | | SMP | | SIP | | TIP (unassigned) | — | SSP | SPUA-A/B |
| 0000–0FFF | 8000–8FFF | 10000–10FFF | 18000–18FFF | 20000–20FFF | 28000–28FFF | | | E0000–E0FFF | *15: SPUA-A* F0000–FFFFF |
| 1000–1FFF | 9000–9FFF | 11000–11FFF | | 21000–21FFF | 29000–29FFF | | | | |
| 2000–2FFF | A000–AFFF | 12000–12FFF | | 22000–22FFF | 2A000–2AFFF | | | | |
| 3000–3FFF | B000–BFFF | 13000–13FFF | 1B000–1BFFF | 23000–23FFF | 2B000–2BFFF | | | | *16: SPUA-B* 100000–10FFFF |
| 4000–4FFF | C000–CFFF | 14000–14FFF | | 24000–24FFF | 2C000–2CFFF | | | | |
| 5000–5FFF | D000–DFFF | | 1D000–1DFFF | 25000–25FFF | 2D000–2DFFF | | | | |
| 6000–6FFF | E000–EFFF | 16000–16FFF | 1E000–1EFFF | 26000–26FFF | 2E000–2EFFF | | | | |
| 7000–7FFF | F000–FFFF | 17000–17FFF | 1F000–1FFFF | 27000–27FFF | 2F000–2FFFF | | | | |

| | A | ℵ | 好 | 不 |
|---|---|---|---|---|
| Code point | U+0041 | U+05D0 | U+597D | U+233B4 |
| UTF-8 | 41 | D7 90 | E5 A5 BD | F0 A3 8E B4 |
| UTF-16 | 00 41 | 05 D0 | 59 7D | D8 4C DF B4 |
| UTF-32 | 00 00 00 41 | 00 00 05 D0 | 00 00 59 7D | 00 02 33 B4 |

# Strings

## *char* vs. *wchar_t*

- The *char* type was the original character type in C and C++.
  - The *char* type can be used to store characters from
    - the ASCII character set
    - any of the ISO-8859 character sets
    - individual bytes of multi-byte characters such as UTF-8 encoding of the Unicode character set.
  - In the Microsoft compiler, *char* is an 8-bit type.
- The *wchar_t* type is a wide character type, intended to be capable of representing any valid code point in a single integer.
  - The size of *wchar_t* is compiler- and system-specific.
    - In Microsoft compiler, it represents a 16-bit wide character.
- There are *char8_t*, *char16_t*, and *char32_t* to present 8-bit, 16-bit, and 32-bit wide characters.

C++ Copy

```
char     ch1{ 'a' };  // or { u8'a' }
wchar_t  ch2{ L'a' };
char16_t ch3{ u'a' };
char32_t ch4{ U'a' };
```

# Reference


- https://docs.microsoft.com/en-us/cpp/cpp/char-wchar-t-char16-t-char32-t?view=msvc-160
- https://www.cplusplus.com/reference/stl/
- https://www.cplusplus.com/doc/oldtutorial/templates/
- https://ecomputertips.com/what-is-cache-memory/
- https://www.2brightsparks.com/resources/articles/introduction-to-hashing-and-its-uses.html#:~:text=Hashing%20is%20an%20algorithm%20that,which%20represents%20the%20original%20string.&text=A%20hash%20is%20usually%20a%20hexadecimal%20string%20of%20several%20characters.
- https://en.wikipedia.org/wiki/Hash_function
- https://en.wikipedia.org/wiki/Standard_Template_Library
- https://www.boost.org/