CS601: Software Development for Scientific Computing

Autumn 2023

Week11: Intermediate C++, Structured Grids

References and Const

- We saw reference variables earlier
- Closely related to pointers:
  - Directly name another object of the *same* type.
  - A pointer is defined using the * (dereference operator) symbol. A reference is defined using the & (address of operator) symbol. Furthermore, unlike in pointer definitions, a reference must be defined/initialized with the object that it names *(cannot be changed later)*.

References

```
int n=10;
int &re=n; //re must be initialized
int* ptr; //ptr need not be initialized here
ptr=&n //ptr now initialized (now pointing to n)
int x=20;
ptr=&x; //ptr now pointing to x
re=x; //doesn’t do what you think. Re still points to n but the value of n is changed..
printf(“%p %p\n”,&re, &n); // re and n are naming the same box in memory. Hence, they have the same address.
```

const

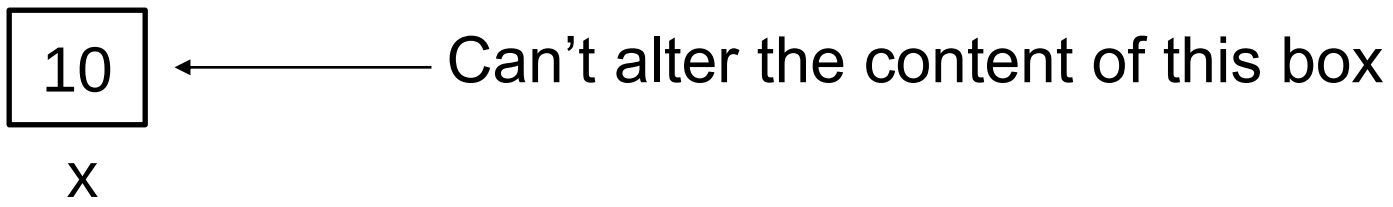
- A type qualifier
- The type is a constant (cannot be modified).
- const is the keyword
- Example:

```
const int x=10; //equivalent to: int const x=10;
//x is a constant integer. Hence, cannot be
//modified.
```

In what memory segment does x gets stored?

Const Properties

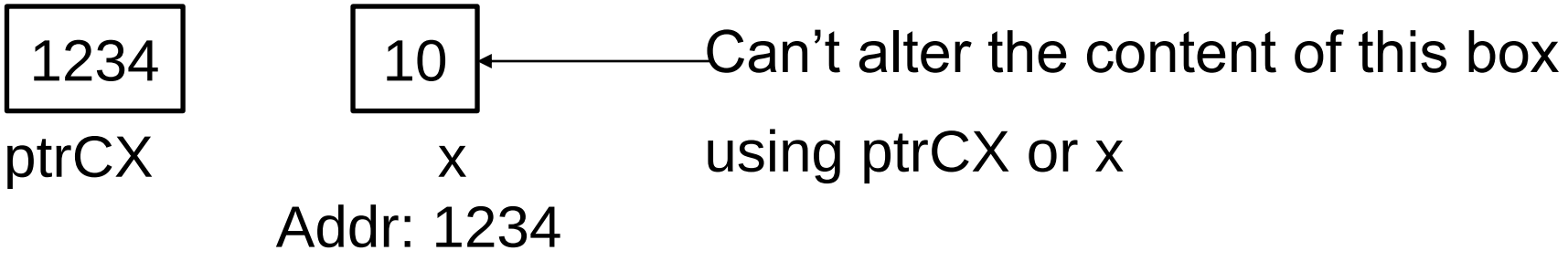
- Needs to be initialized at the time of definition
- Can’t modify after definition
- ```
  const int x=10;
  x=20; //compiler would throw an error
  ```
- ```
  int const x=10;
  x=10; //can’t even assign the same value
  ```
- ```
  int const y; //uninitialized const variable y. Useless.
  ```


  ```

Const Example1 (error)

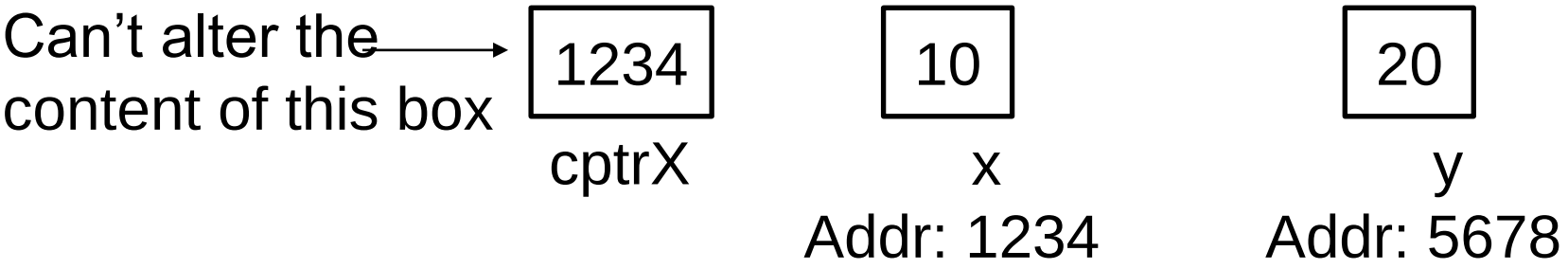
```
/*ptrCX is a pointer to a constant integer. So, can't
modify what ptrCX points to.*/
const int* ptrCX; //or equivalently:
int const* ptrCX;

int const x=10;
ptrCX = &x;
*ptrCX = 20; //Error
```



Const Example2 (error)

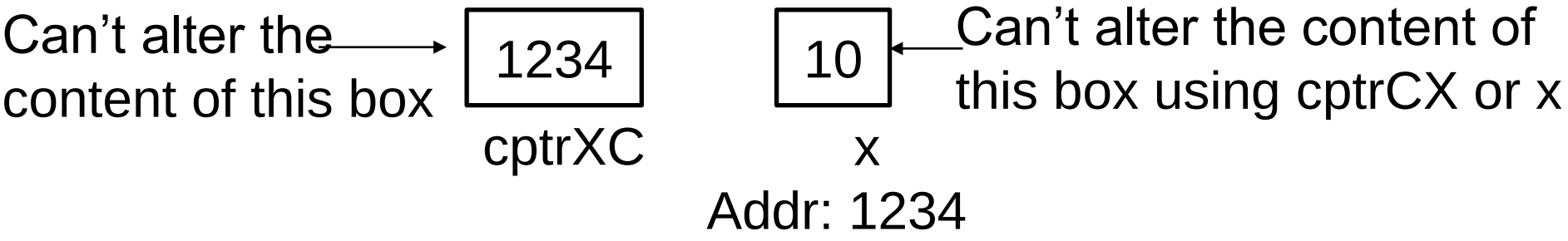
```
/*cptrX is a constant pointer to an integer. So, can't
point to anything else after initialized.*/
int x=10, y=20;
int *const cptrX=&x;
cptrX = &y; //Error
```



Const Example3 (error)

```
/*cptrXC is a constant pointer to a constant integer. So,
can't point to anything else after initialized. Also,
can't modify what cptrXC points to.*/

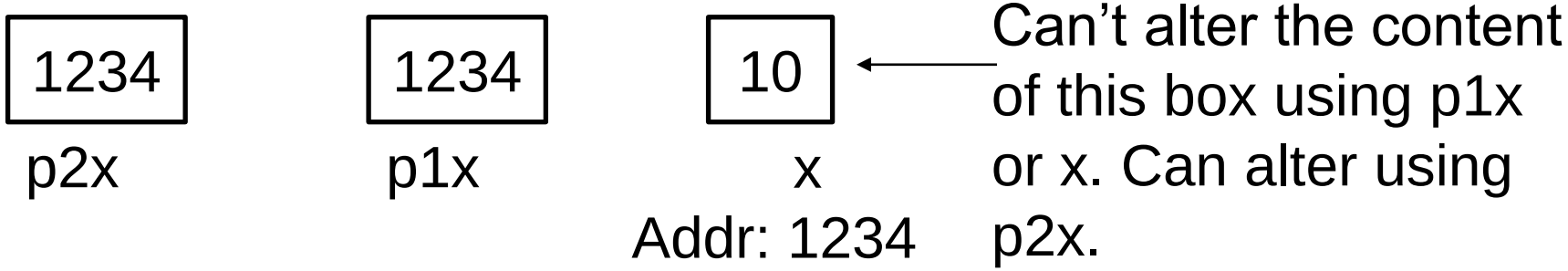
const int x=10, y=20;
const int *const cptrXC=&x;
int const *const cptrXC2=&x; //equivalent to prev. defn.
cptrXC = &y; //Error
*cptrXC = 40; //Error
```



Const Example4 (warning)

```
/*p2x is a pointer to an integer. So, we can use p2x to
alter the contents of the memory location that it points
to. However, the memory location contains read-only data -
cannot be altered. */

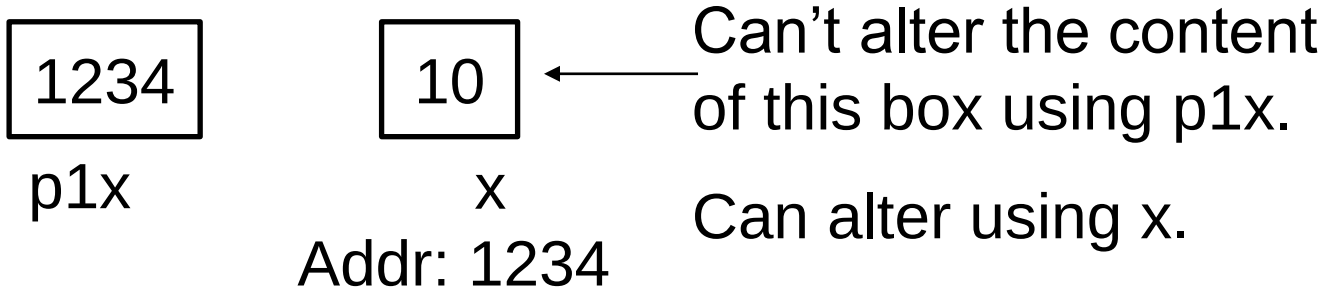
const int x=10;
const int *p1x=&x;
int *p2x=&x; //warning
*p2x = 20; //goes through. Might crash depending on memory
location accessed
```



Const Example5 (no warning, no error)

```
/*p1x is a pointer to a constant integer. So, we can't use
p1x to alter the content of the memory location that it
points to. However, the memory location it points to can
be altered (through some other means e.g. using x)*/

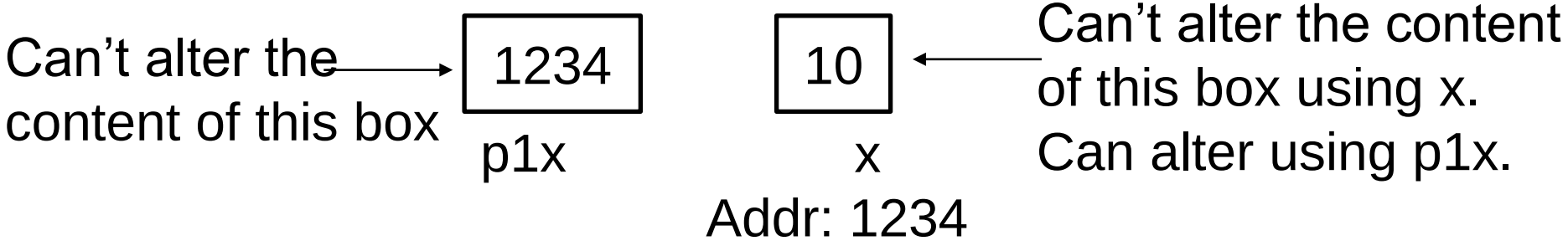
int x=10;
const int *p1x=&x;
```



Const Example6 (warning)

```
/*p1x is a constant pointer to an integer. So, we can use
p1x to alter the contents of the memory location that it
points to (and we can't let p1x point to something else
other than x). However, the memory location contains read-
only data - cannot be altered. */

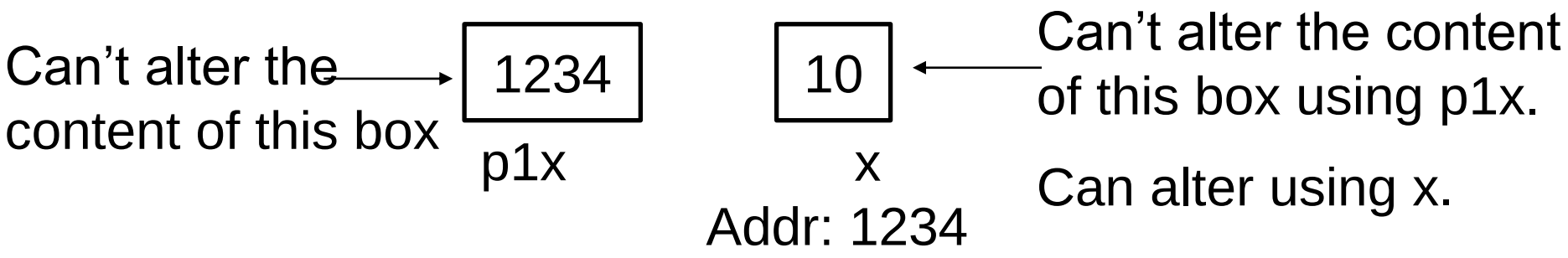
const int x=10;
int *const p1x=&x;//warning
*p1x = 20; //goes through. Might crash depending on memory
location accessed
```



Const Example7 (no warning, no error)

```
/*p1x is a constant pointer to a constant integer. So, we
can't use p1x to alter the content of the memory location
that it points to. However, the memory location it points
to can be altered (through some other means e.g. using
x)*/

int x=10;
const int *const p1x=&x;
```



Const and References - Summary

- Allow for compiler optimizations
  - pass-by-reference: allows for passing large objects to a function call
- Tell us immediately (by looking at the interface) that a parameter is read-only

Templating Functions and Classes

- Provide a recipe for generating multiple versions of the function/class based on the data type of the data on which the function/class operates upon

- Demo:

Computing Dot Product (in Week3)

```
#include"scprod_v4.h"
double cs601::ddot(int dim, double vec1[], double vec2[]) {
        double result=0.;
        for(int i=0;i<dim;i++) {
                result += (vec1[i] * vec2[i]);
        }
        return result;
}
```

What if user wants to compute dot product with integers?

Computing Dot Product (in Week3)

```
#include"scprod_v4.h"
double cs601::ddot(int dim, double vec1[], double vec2[]) {
        double result=0.;
        for(int i=0;i<dim;i++) {
                result += (vec1[i] * vec2[i]);
        }
        return result;
}

int cs601::ddot(int dim, int vec1[], int vec2[]) {
        int result=0.;
        for(int i=0;i<dim;i++) {
                result += (vec1[i] * vec2[i]);
        }
        return result;
}
```

Computing Dot Product (in Week11)

```
template<typename T>
T cs601::scprod(int dim, T* vec1, T* vec2) {
        T result=0.;
        for(int i=0;i<dim;i++) {
                result += (vec1[i] * vec2[i]);
        }
        return result;
}
```

Should we put this code in a .h file or .cpp file?

Computing Dot Product (in Week11)

```
double* vector1, *vector2;
//allocate memory for vectors
vector1 = new double[dim];
vector2 = new double[dim];

//initialize vectors with some double values
for(int i=0;i<dim;i++) {
        vector1[i] = 1.1;
        vector2[i] = 1.1;
}

//multiply vectors of doubles and store the result in a new double:
double dResult=0.;
dResult = cs601::scprod<double>(dim, vector1, vector2);
```

Computing Dot Product (in Week11)

```
int* vector3, *vector4;
//allocate memory for vectors
vector3 = new int[dim];
vector4 = new int[dim];

//initialize vectors with some int values
for(int i=0;i<dim;i++) {
        vector3[i] = i+1;
        vector4[i] = i+1;
}
int iResult=0.;
//multiply vectors of ints and store the result in a new int:
iResult = cs601::scprod<int>(dim, vector3, vector4);
```

Computing Dot Product (in Week11)

```
double dResult=0.;
dResult = cs601::scprod(dim, vector1, vector2);
```

```
int iResult=0.;
//multiply vector of int and store the result in a new int
iResult = cs601::scprod(dim, vector3, vector4);
```

The above also works when `scprod` is a template function. Type resolution must be unambiguous

Class MyVec (in Week10)

```
//CS601: example code used to show class declaration.
#ifndef MYVEC_H
#define MYVEC_H
class MyVec{
        //private attributes
        double* data;
        int vecLen;
public:
        MyVec(){data=0;vecLen=0;}
```

What if user wants to have a MyVec class with integer data?

Class Templates

- Like function templates but for templating classes

Refer to demo example for class and function templates

Standard Template Library (STL)

- Large set of frequently used data structures and algorithms
  - Defined as *parametrized* data types and functions
  - Types to represent complex numbers and strings, algorithms to sort, get random numbers etc.
- Convenient and bug free to use these libraries
- E.g. `vector, map, queue, pair, sort` etc.
- Use your own type only for efficiency considerations - *only if you are sure!*

STL - Motivation

Real-world view
source:wikipedia

Coconut meat, raw

Nutritional value per 100 g (3.5 oz)	
Energy	354 kcal (1,480 kJ)
Carbohydrates	15.23 g
Sugars	6.23 g
Dietary fiber	9.0 g
Fat	33.49 g
Saturated	29.698 g
Monounsaturated	1.425 g
Polyunsaturated	0.366 g
Protein	3.33 g
Tryptophan	0.039 g
Threonine	0.121 g
Isoleucine	0.131 g
Leucine	0.247 g
Lysine	0.147 g
Methionine	0.062 g
Cystine	0.066 g
Phenylalanine	0.169 g
Tyrosine	0.103 g
Valine	0.202 g
Arginine	0.546 g
Histidine	0.077 g
Alanine	0.170 g
Aspartic acid	0.325 g
Glutamic acid	0.761 g
Glycine	0.158 g
Proline	0.138 g
Serine	0.172 g
Vitamins	**Quantity** **%DV†**

Consider the nutrients (constituents) present in edible part of coconut.
How would you capture the Real-world view in a Program?

```
vector<pair<string, float> > constituents;
```

Container

- Holder of a collection of objects
- Is an object itself
- Different types:
  - sequence container
  - associative container (ordered/unordered)
  - container adapter

Sequence Container

- Provide fast sequential access to elements
- Factors to consider:
  - Cost to add/delete an element
  - Cost to perform non-sequential access to elements

container name	**comments**
`vector`	Flexible array, fast random access
`string`	Like `vector`. Meant for sequence of characters
`list/slist`	doubly/singly linked list. Sequential access to elements (bidirectional/unidirectional).
`deque`	Double-ended queue. Fast random access, Fast append
`array`	Intended as replacement for ‘C’-style arrays. Fixed-sized.

Container Adapter

- Provide an interface to sequence containers
  - `stack, queue, priority_queue`

Associative Container

- Implement sorted data structures for efficient searching (O(log n)) complexity.
  - Set, map, multiset, multimap

container name	comments
`set`	Collection of unique sorted keys. Implemented as class template
`map`	Collection of key-value pairs sorted by unique keys. Implemented as class template

Unordered Associative Container

- Implement hashed data structures for efficient searching (O(1) best-case, O(n) worst-case complexity).
  - `unordered_set, unordered_map, unordered_multiset, unordered_multimap`

Recap: Returning References-Example1

- How can we assign one object to another?

```
Apple a1("Apple", 1.2); //constructor Apple::Apple(string, float)
                        //is invoked
Apple a2; //constructor Apple::Apple() is invoked.
a2  = a1 //object a1 is assigned to a2;assignment operator is invoked
```

```
Apple& Apple::operator=(const Apple& rhs)
```

Called Copy Assignment Operator

```
Apple& Apple::operator=(const Apple& rhs) {
commonName = rhs.commonName;
weight = rhs.weight;
energyPerUnitWeight = rhs.energyPerUnitWeight;
constituents = rhs.constituents;
return *this;
}
```

What is Move Assignment Operator?

this

- Implicit variable defined by the compiler for every class
  - E.g. `MyVec *this;`
- All member functions have `this` as an implicit first argument
  - E.g.

```
int MyVec::GetVecLen() const;
```

would actually be:

```
int MyVec::GetVecLen(MyVec* this) const;
```

Overloading +=

- `MyVec v1;`
  `v1+=3;`
- `MyVec& MyVec::operator+=(double)`

Overloading +=

- `MyVec v1;`

  `v1+=3;`
  - `MyVec& MyVec::operator+=(double)`
- `MyVec v2;`

  `v2+=v1;`
  - `MyVec& MyVec::operator+=(const MyVec& rhs)`
  - What if you make the return value above `const`?

    Disallow: `(v2+=v1)+=3;`

Overloading +

- `v1=v1+3;`

  Single-argument constructors: *allow implicit conversion from a particular type to initialize an object.*

  – `const MyVec MyVec::operator+(double val)`
- `v3=v1+v2;`

  1. `const MyVec MyVec::operator+(const MyVec& vec2) const;`

  OR

  2. `friend const MyVec operator+(const MyVec& lhs, const MyVec& rhs);`

v1=3+v1 is compiler error! Why?

Operator Overloading - Guidelines

- If a binary operator accepts operands of different types and is commutative, both orders should be overloaded
- Consistency:
  - If a class has ==, it should also have !=
  - += and + should result in identical values
  - define your copy assignment operator if you have defined a copy constructor

Matrix Algebra and Efficient Computation

- **Pic source: the Parallel Computing Laboratory at U.C. Berkeley: A Research Agenda Based on the Berkeley View (2008)**

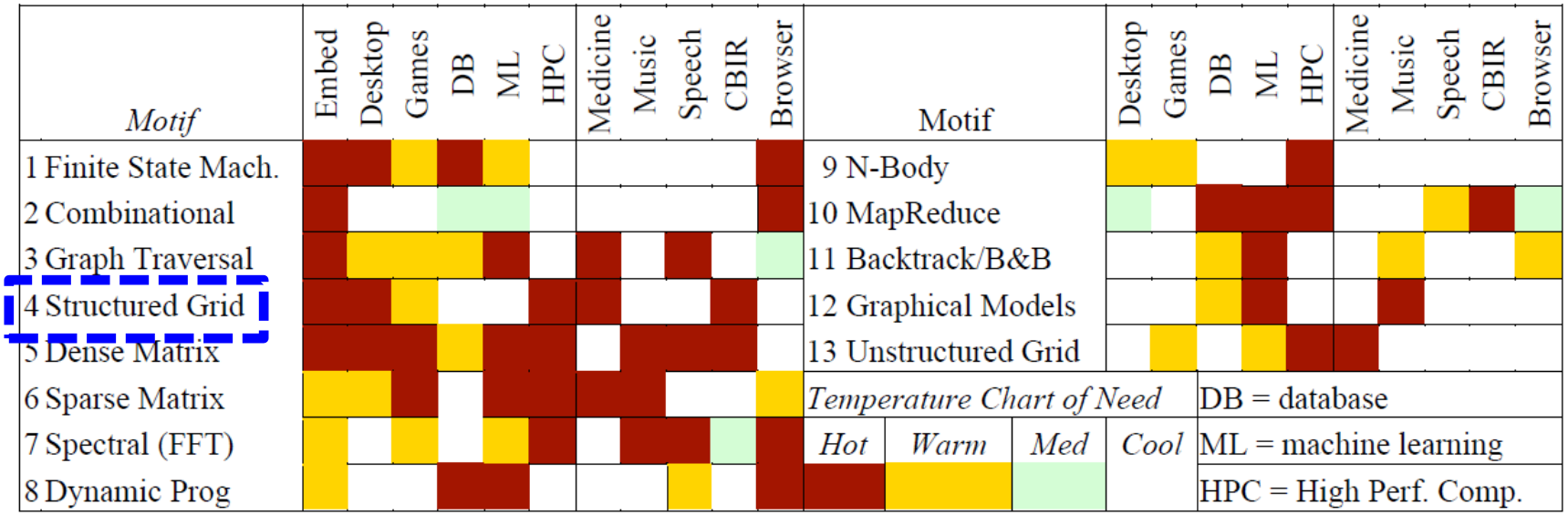


Figure 4. Temperature Chart of the 13 Motifs. It shows their importance to each of the original six application areas and then how important each one is to the five compelling applications of Section 3.1. More details on the motifs can be found in (Asanovic, Bodik et al. 2006).

Discretization

- Cannot store/represent infinitely many continuous values
  - To model turbulent features of flow through a pipe, say, I am interested in `velocity` and `pressure` at all `points` in a region of interest
    1. Represent region of interest as a mesh of small discrete cells - ***discretization spacing***
    2. Solve equations for each cell

Example:

```
diameter of the pipe = 5cm
length=2.5cm
discretization spacing = 0.1mm
(volume of cylinder = πr²h)
```

Exercise: *how many variables do you need to declare?*

Discretization

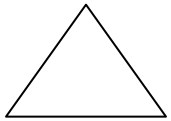
- All problems with ‘continuous’ quantities don’t require discretization
  - Most often they do.
- When discretization is done:
  - How refined is your discretization depends on certain parameters: step-size, cell shape and size. E.g.
    - Size of the largest cell (PDEs in FEM),
    - Step size in ODEs
  - Accuracy of the solution is of prime concern
    - Discretization always gives an approximate solution. Why?
    - Errors may creep in. Must provide an estimate of error.

Accuracy

- Discretization error
  - Is because of the way discretization is done
  - E.g. use more number of rays to minimize discretization error in ray tracing
- Solution error
  - The equation to be solved influences solution error
  - E.g. use more number of iterations in PDEs to minimize solution error
- Accuracy of the solution depends on both solution and discretization errors
- Accuracy also depends on cell shape

Cell Shape

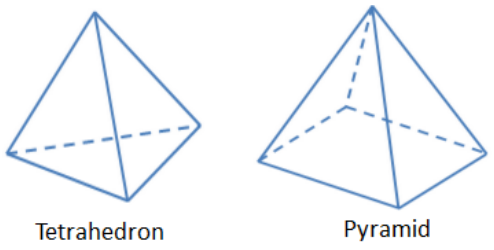
- 2D:



triangle

quadrilateral

- 3D: triangular or quadrilateral faced. E.g.



source: wikipedia

Tetrahedron: 4 vertices, 4 edges, 4△ faces
Pyramid: 5 vertices, 8 edges, 4 △ and 1 ▭ face
Triangular prism: 6 vertices, 9 edges, 2△ and 3▭ faces
Hexahedron: 8 vertices, 12 edges, 6 ▭ faces