Imperial College London

Advanced Programming

Pointers

Dr Adriana Paluszny

[Some of the slides are adapted from the original Bjarne Stroustrup slides on C++]

Imperial College London

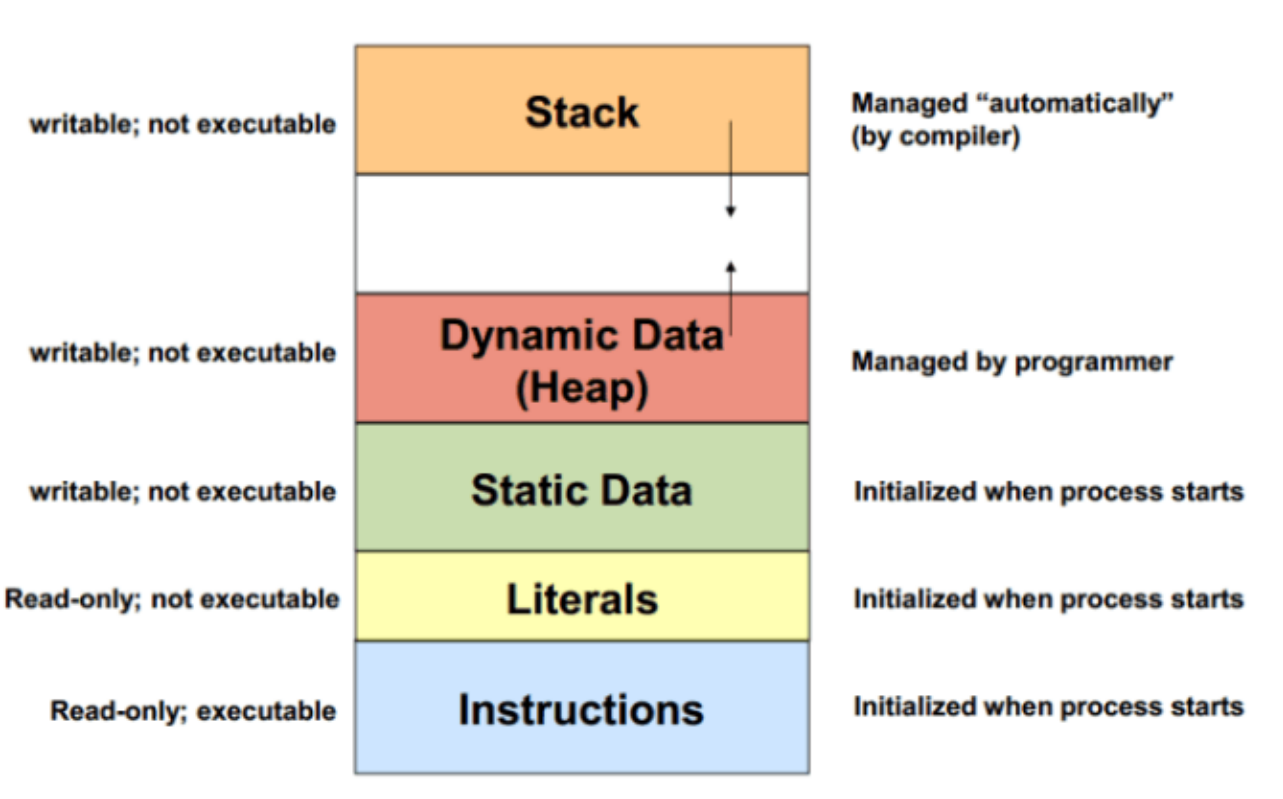
Memory

- C++ provides several mechanisms for memory management, including automatic, dynamic, and static memory allocation.
- **Automatic memory** allocation happens when a variable is declared, and the memory is allocated on the stack (local variables, stored in ‘stack’).
- **Dynamic memory** allocation happens when memory is allocated during runtime using the new operator, and deallocated using the delete operator (stored in ‘heap’).
- **Static memory** is allocated at compile time, and the memory is deallocated automatically when the program terminates (stored in RAM next to program data, not stack)

Imperial College London

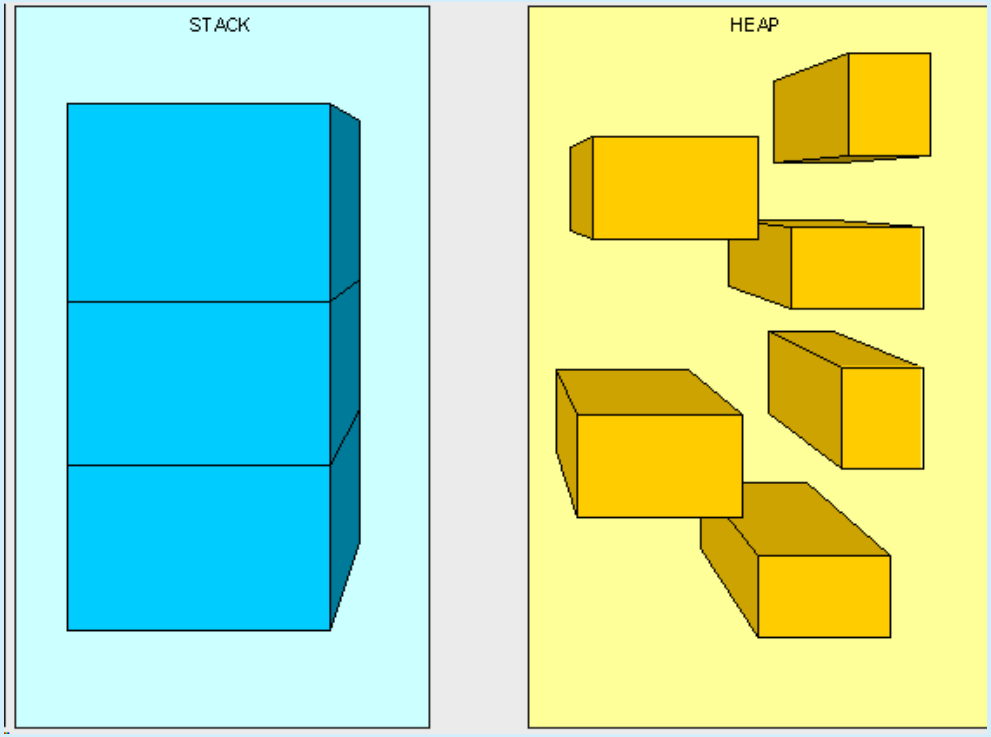
Memory

This is only an illustration, these bits can be in different order



Imperial College
London

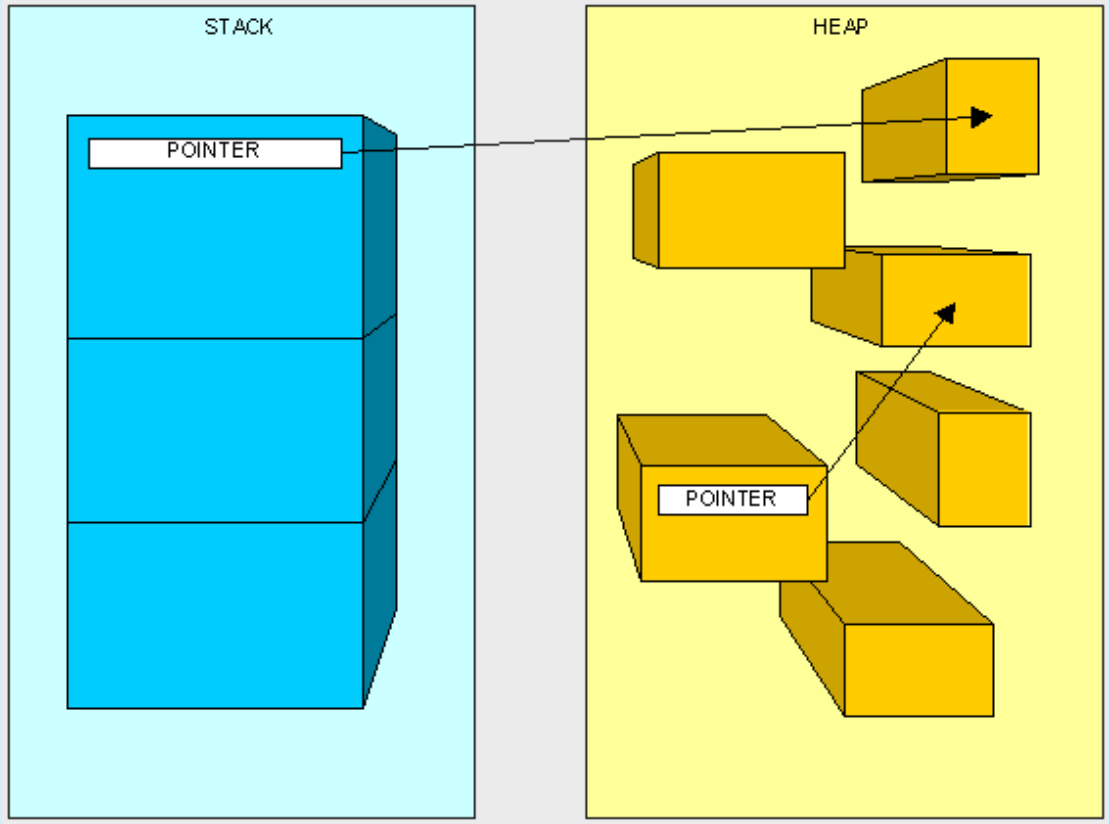
Stack & Heap



Imperial College London

Stack & Heap

- Allocation by the compiler
- Faster
- Memory never gets fragmented
- Cannot resize variables
- Objects automatically destroyed after use
- Usually 1MB



- Allocation by the user
- Slower
- Memory gets fragmented over time
- Variables can be resized
- Objects need to be deallocated by user
- Default size is 8MB or 1/64th of the physical memory within the 8 Mb to 1 Gb range

Imperial College London

Pointers

```
int a = 10;             //initializes ‘a’ to 10

int *b = &a;
*b = 20;                //changes the value of ‘a’ to 20

int **c = &b;
**c = 30;               //changes the value of ‘a’ to 30
```

Passing Pointers to Functions

- Functions can have pointers as parameters
  - This can be used be used to get the function to return more than one value, but in C++ this is done more easily by passing a variable by reference (see next section)
    - By giving a pointer to a variable, the function can write a value to the memory location indicated by the pointer and thus return an extra value.

```
int my_function(int *b);
int* my_other_function(double f);
```

  - This is also a useful way for a function to take in an array (remembering the equivalence between arrays and pointers)

Imperial College London

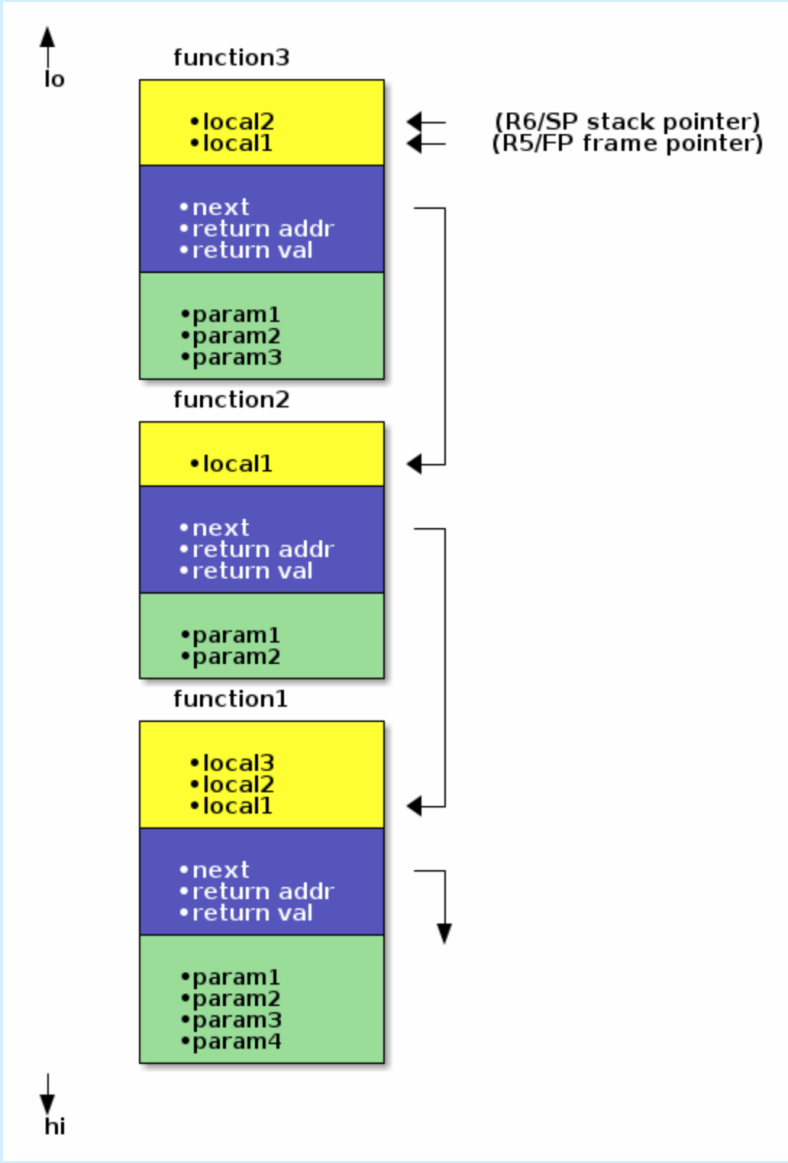
Stack

- **Last-In-First-Out (LIFO):** The stack is just like a stack of plates
- **Automatic memory management:** managed automatically by the program's runtime environment. Memory is allocated and deallocated as function calls are made and returned. A “stack frame” is created every time you enter a function. It stores the values of variables passed in/out and local variables
- **Fixed size:** The stack is of a fixed size (set by the OS), on Windows the default stack size is 1MB, on Linux 8MB. This limits how many nested functions you could call, or the size/number of local variables you can store
- **Constant-time operations:** Operations on the stack, such as pushing and popping elements, can be performed in constant time.
- “**Stack overflow error**” is when the stack doesn’t have any more space left

Imperial College London

Stack Frame

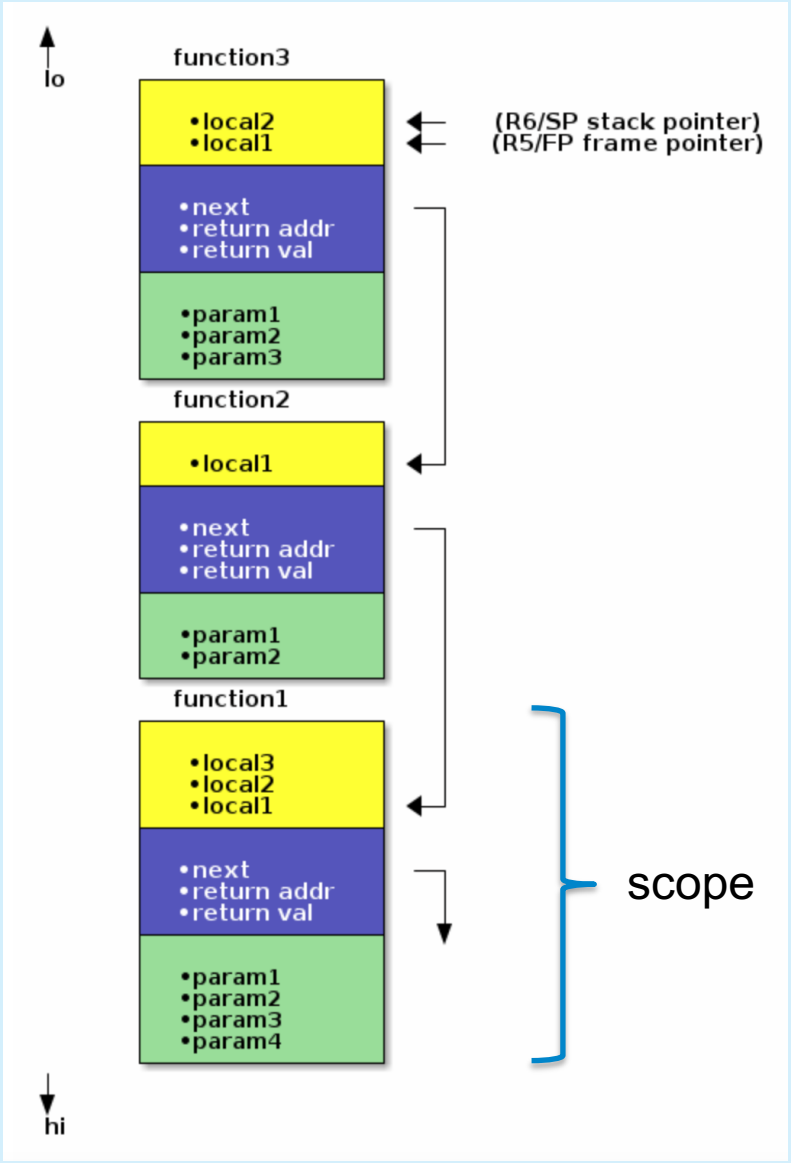
- Every time a function is called, a new “stack frame” is created on top of the call stack to store information about the function call
- This includes
  - space for the parameters
  - space to hold the return address/value and other book keeping value(s)
  - space for the local variables
- When the function returns, the stack frame is removed from the top of the stack, and program execution resumes at the return address.



Imperial College London

Scoping

- We've seen previously a discussion on "scope" of variables
- This is how long they "live"
- It's easy to see why variables don't live beyond their function when you think about stack frames
- Once your program exists main(), the OS then handles deletion of all the memory your program used, even if you leaked memory



Imperial College London

Heap

- **Dynamic memory allocation: t**he heap allows programs to allocate and deallocate memory at runtime.
- **Unordered memory:** memory blocks can be allocated and deallocated in any order, no guarantee that they will be physically adjacent to each other.
- **Random access**: memory blocks in the heap can be accessed randomly, which means that the program can access any block of memory at any time.
- **Memory fragmentation**: the heap can become fragmented, meaning that there are small gaps of unused memory between allocated blocks. This can reduce the amount of usable memory in the heap.
- **“Memory leaks”:** when a program does not deallocate memory when it is no longer needed, a memory leak can occur. This means that the program continues to hold onto memory that it is no longer using, which can lead to performance problems and crashes.

Pointers

- Pointers are something that Python doesn't have at all
- A pointer is a memory address
- Pointers are declared by putting a star in front of the variable name.
  - e.g. a pointer to a memory location that contains an integer would be declared as follows: `int *P1;`
- The pointer (i.e. memory location) of any variable can be obtained using the & symbol
  - e.g. If `A` is a simple variable, then `&A` is the pointer to the memory location where `A` is stored
- Conversely, if you have a pointer, the `*` symbol can be used to access the value stored at that memory location
  - e.g. If `P1` is a pointer to an integer, then `*P1` is the integer stored at the memory location `P1`

Imperial College London

How are pointers stored?

- In the C++ programming language, the memory size of a pointer depends on the architecture of the system
- On a 32-bit system, a pointer typically occupies 4 bytes (32 bits) of memory.
- On a 64-bit system, a pointer usually occupies 8 bytes (64 bits) of memory.

- Limits memory usage:
  - In a 32-bit program, you can only address 4 Gb of RAM - and will depend on the operating system (could be less!).
  - In a 64-bit program, the theoretical limit is enormous (16 exabytes), but practical limitations place the maximum at several terabytes, primarily driven by hardware and operating system constraints.

The significant shift from 32-bit to 64-bit computing occurred gradually over the mid-2000s, driven by advancements in hardware technology and increasing memory demands of software applications.

An example with simple pointers

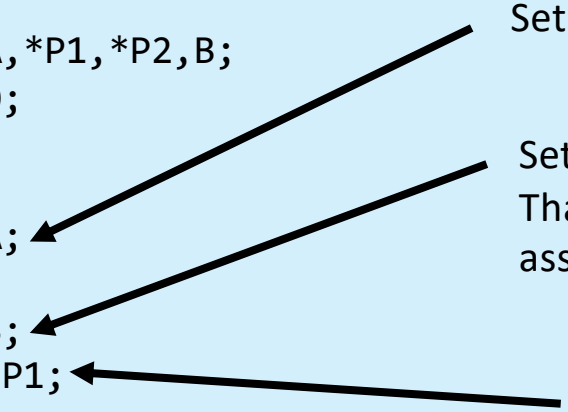
```
#include <iostream>
using namespace std;


void main(void)
{
   int A,*P1,*P2,B;
   A=500;


   P1=&A;

   P2=&B;
   *P2=*P1;


   cout << "B: " << B << endl;
}
```



Sets P1 equal to the memory location where A is stored

Sets P2 equal to the memory location where B is stored. That B hasn't been given a value is immaterial as declaring it assigns memory to the variable.

Sets the value stored at memory location P2 equal to the value stored at memory location where P1. This has the effect of setting B equal to A.

Pointers and Memory Allocation

- The previous example might seem a bit esoteric and not very useful
  - It will become very important later when we look at advanced memory structure such as linked lists and trees
- In the meantime, there is a very useful thing that can be done using pointers:
  - Pointers can have memory directly allocated to them, rather than simply pointing at an existing variable
  - This means that we can effectively have arrays where you don't need to know the size of the array when writing the program, but can set its size based on user input and/or calculated values

Allocating Memory

- Memory allocation is done using the new command
  - It can also be done using malloc, though this is considered to be more of a C, rather than C++ method
- Any memory that is allocated should be de-allocated using delete
- To allocate memory for a single variable:

e.g. To allocate memory for a single integer:

```
int *P;
P=new int;
*P=24;
delete P;
```

- Memory can also be allocated to a pointer to be used like an array:

e.g. For a pointer containing enough memory to store 20 integers:

```
int *P;
P=new int[20];
P[12]=24;
delete[] P;
```

Dynamic arrays are stored in the heap

Imperial College London

The first item of an array

The name of an array variable is also a pointer to the memory location of the first item in the array

–i.e. If `A` is an array (e.g. `int A[20]`), then `A≡&A[0]`

```
int a;
char A[20];
```

Static arrays are stored in the stack

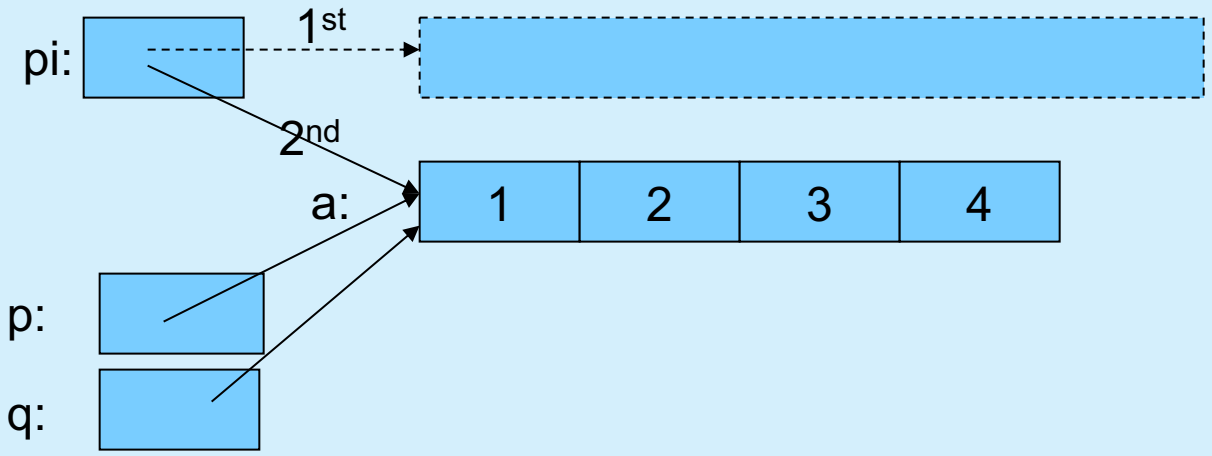
```
void f(int n)
{
      int b;
      int* p = &b;      // pointer to individual variable
      p = &a;           // now point to a different variable
      char* pc = A;     // the name of an array names a pointer to its first element
      pc = &A[0];       // equivalent to pc = ac
      pc = &A[n];       // pointer to ac's nth element (starting at 0th)
                        // warning: range is not checked
      // …
}
```

Imperial College London

Arrays (often) convert to pointers

```
void f(int pi[ ])     // equivalent to void f(int* pi)
 {
    int a[ ] = { 1, 2, 3, 4 };
    int b[ ] = a;     // error: copy isn't defined for arrays
    b = pi;           // error: copy isn't defined for arrays. Think of a
                      // (non-argument) array name as an immutable pointer
    pi = a;           // ok: but it doesn't copy: pi now points to a's first element
                      // Is this a memory leak? (maybe)
    int* p = a;       // p points to the first element of a
    int* q = pi;      // q points to the first element of a
}
```

Static arrays are stored in the stack



Imperial College London

Pointers

- You can perform arithmetic operations on pointers!

```
int a[5] = {1, 2, 3, 4, 5};
int *p = a; // Pointer to the beginning of the array

// Accessing elements using pointer arithmetic
std::cout << "Elements of the array: ";
for (int i = 0; i < 5; ++i) // Accessing elements
 std::cout << p[i] << ”\n";

// Adding 2 to the pointer
// move it two positions forward in the array
p += 2;
std::cout << "After adding 2: " << *p << std::endl;
// This should print the value 3
```

Imperial College London

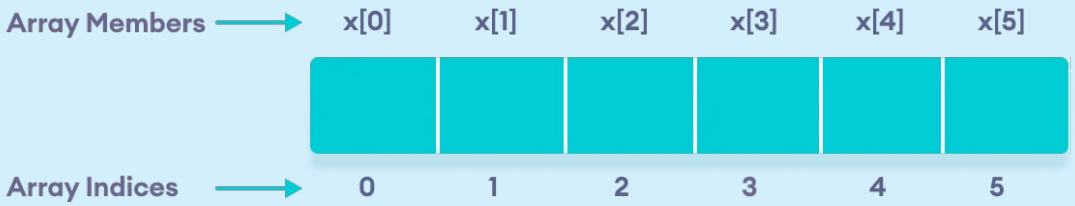
Arrays and Pointers

- Arrays are closely related to pointers and can be though of as being a pointer with pre-allocated memory (there are subtle differences, but not enough to worry about):
  - Internally they are treated slightly differently as arrays are allocated within the stack (a smallish piece of memory automatically assigned to a program by the operating system), while dynamically allocated memory is allocated within the heap
  - Dynamically allocated variables can thus be used to allocate essentially any memory (including virtual memory) available to the operating system, while the total size of static arrays that you can use in a program is far more limit
    - In Windows the default stack size is 1MB (though it can be changed)

Imperial College London

Declaration and Initialization of an Array

```
// declare and initialize and array
int x[6] = {19, 10, 8, 17, 9, 15};
```




```
// store only 3 elements in the array
int x[6] = {19, 10, 8};
```

Imperial College London

Why Arrays?

- They are inherently a feature from C
  - There is a lot of C code “out there”
    - Here “a lot” means N*1B lines
  - There is a lot of C++ code in C style “out there”
    - Here “a lot” means N*100M lines
  - You’ll eventually encounter code full of arrays and pointers
- They represent primitive memory in C++ programs
  - We need them (mostly on free store allocated by **new**) to implement better container types
- Avoid arrays whenever you can
  - They are the largest single source of bugs in C and (unnecessarily) in C++ programs
  - They are among the largest sources of security violations, usually (avoidable) buffer overflows

Use `std::vector` instead whenever possible.

Imperial College London

EXERCISE

- Create a static array A of 1000 integers. Fill the new entries of the array with random numbers between 0 and 100. How much memory does your array occupy?
- Create a static array B of 1000 integers. Fill the new entries of the array with random numbers between 1,000,000 and 10,000,000. How much more memory does array B occupy as compared to A?

- In both, make sure to: store input from user to array and print array elements to the screen (or to a file!)

Pointers to Pointers

- Sometimes you might want to use a pointer to a pointer (i.e. the memory location where a memory location's value is stored)
  - As a pointer is indicated by a `*` during its declaration, a pointer to a pointer is indicated by `**`
  - E.g. To define a pointer to a pointer to a memory location containing a `double`:

```
double **P
```

Dynamic Allocation of Higher Dimensional Arrays

- A common use for a pointer to a pointer is to use it as a 2-dimensional array.
- This requires that memory be assigned for an array of pointers
  - Note the type casting that is required as new returns a pointer rather than a pointer to a pointer. E.g if the array is to by `max_i` by `max_j` in size (with both these variables type `int`):

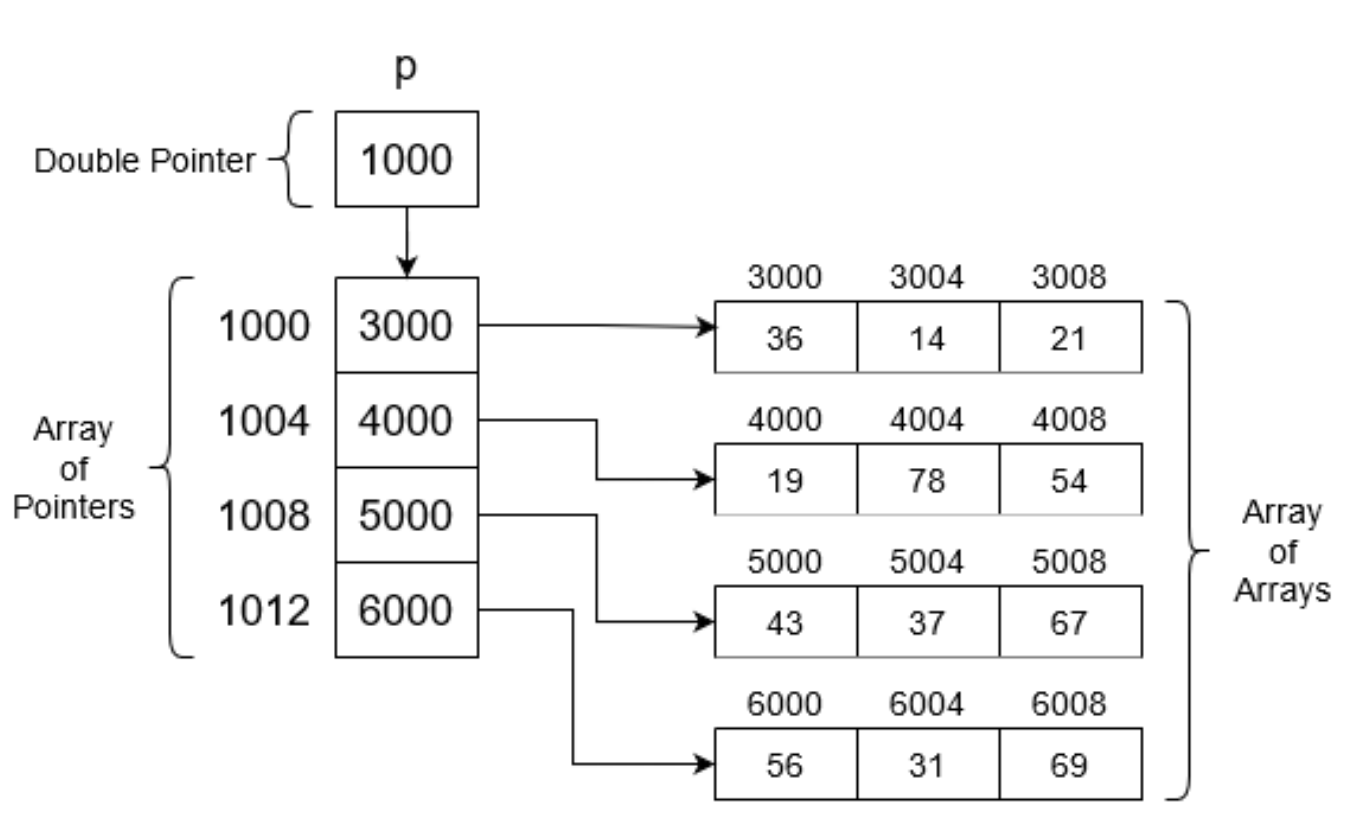
```
P=(double **) new double*[max_i];
```

- Each of entries in the array of pointers now needs memory to be allocated to it
  - Note that by allocating memory in this way means each column need not be the same length, though in this example they are

```
for (i=0;i<max_i;i++) P[i]=new double[max_j];
```

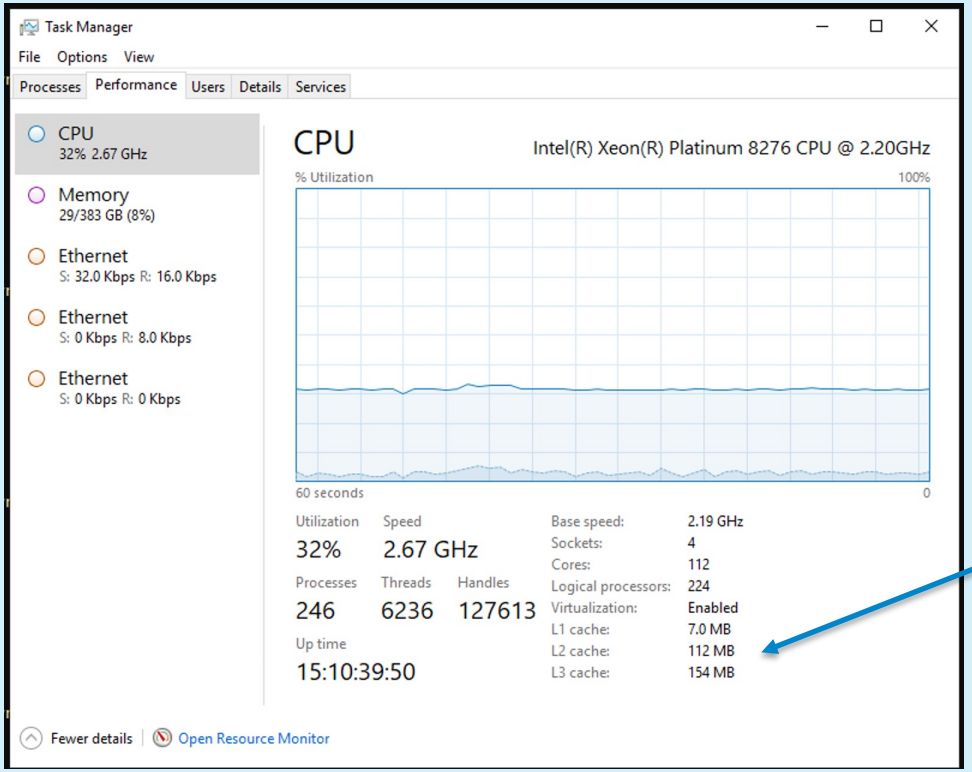
Imperial College London

Double array



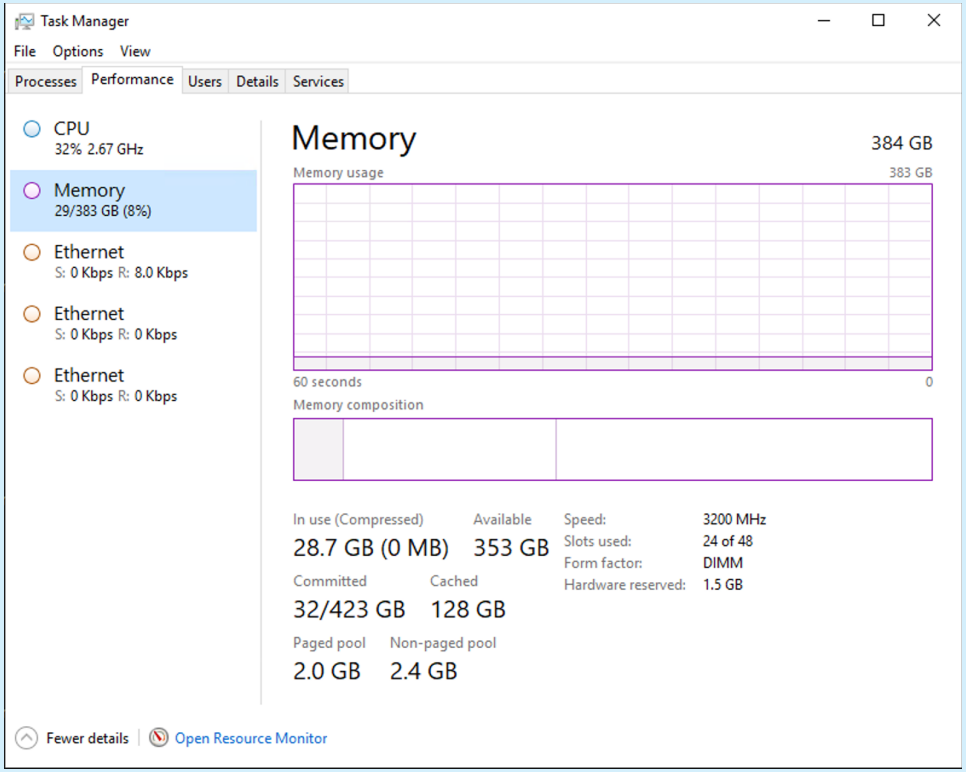
Imperial College London

Task Manager



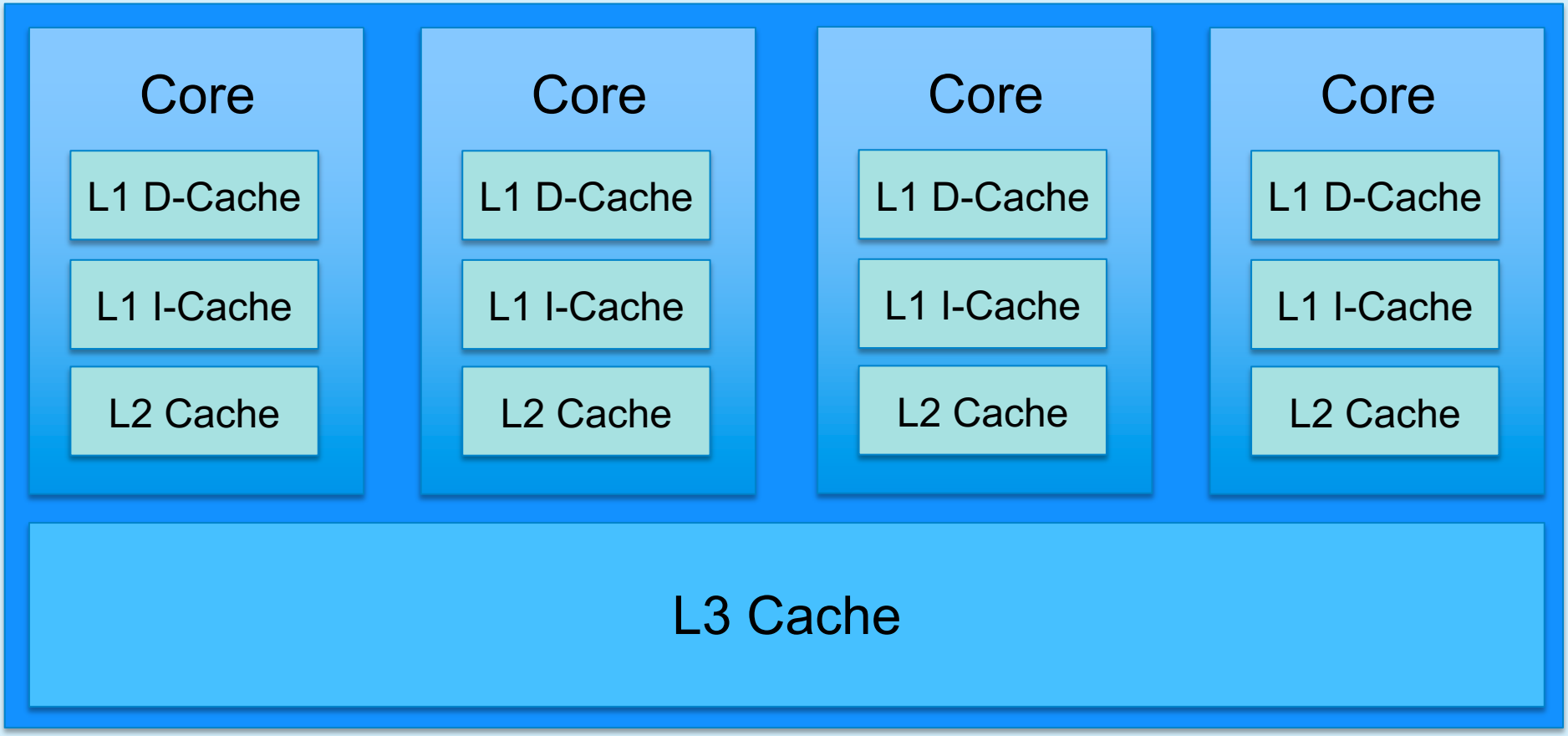
Imperial College London

Task Manager: Memory



Imperial College London

Modern CPUs have a hierarchy of Caches



Imperial College London

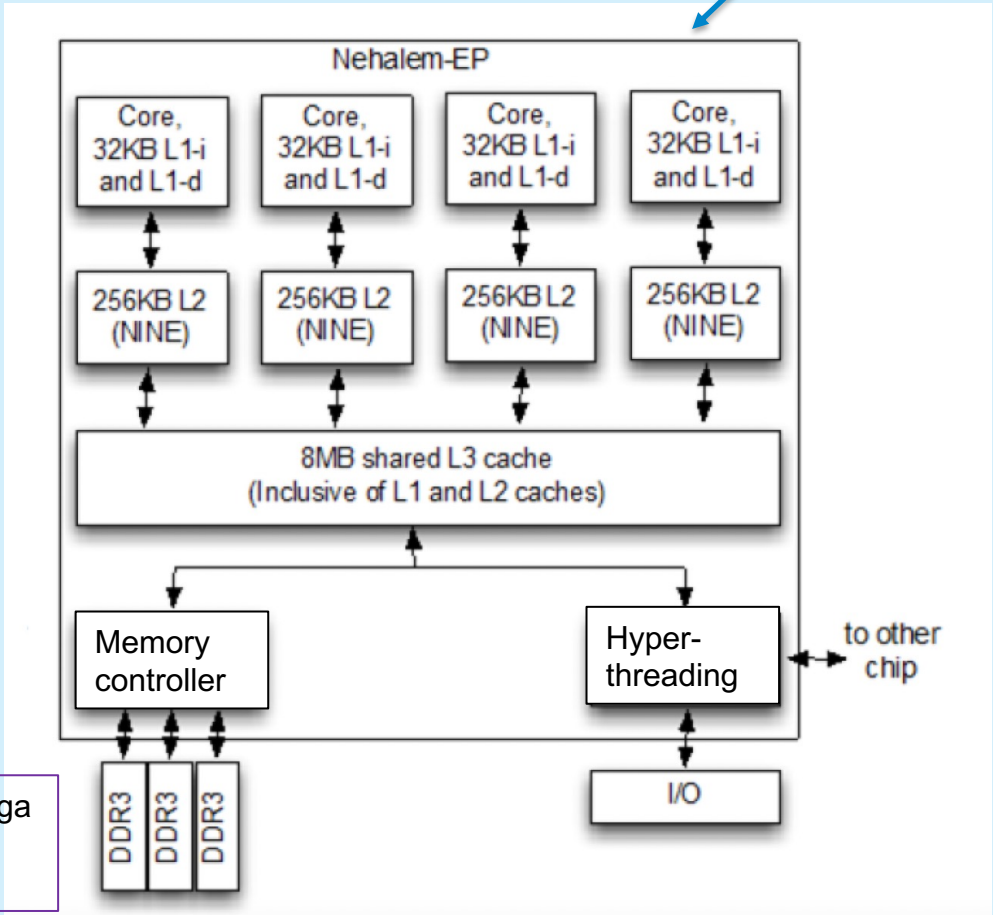
Intel Nehalem Architecture

Faster, smaller

Hyper-threading

NUMA – Non-Uniform Memory Access (different processors accessing memory at different speeds)

DDR4 = 2113 MT/s (mega transfers per second)
Fast memory



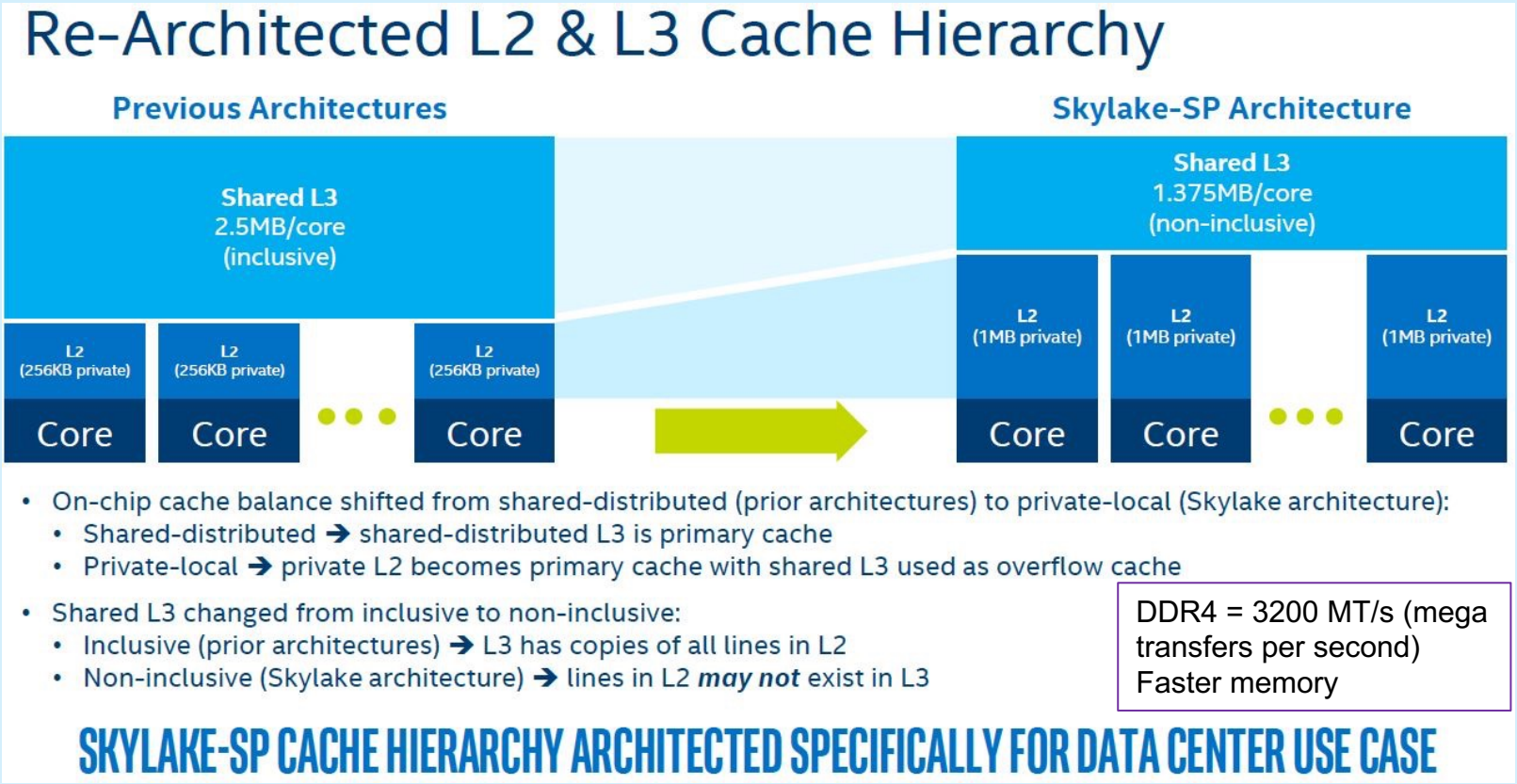
Logo for Core i7 Bloomfield processors

General information	
Launched	November 11, 2008; 14 years ago
Performance	
Max. CPU clock rate	1.06 GHz to 3.33 GHz
Cache	
L1 cache	64 KB per core
L2 cache	256 KB per core
L3 cache	2 MB to 24 MB shared
Architecture and classification	
Technology node	45 nm
Architecture	x86-64
Microarchitecture	Nehalem
Instructions	x86, x86-64
Extensions	MMX, SSE, SSE2, SSE3, SSSE3, SSE4, SSE4.1, SSE4.2 VT-x, VT-d
Physical specifications	
Transistors	731M to 2300M 45 nm
Cores	2-6 (4-8 Xeon)
Socket(s)	LGA 1156 LGA 1366 LGA 1567 μPGA 988
Products, models, variants	
Model(s)	Pentium, Core, Core *i*n and Xeon Series
History	
Predecessor	Core (tock) Penryn (tick)
Successor	Westmere (tick) Sandy Bridge (tock)

Celeron, Pentium Core and Xeon brands

Imperial College London

Skylake Cache Hierarchy – Intel Xeon



DDR4 = 3200 MT/s (mega transfers per second)
Faster memory

For Intel, checkout: https://www.intel.com/content/www/us/en/support/articles/000057727/processors.html

Imperial College London

Intel Architectures

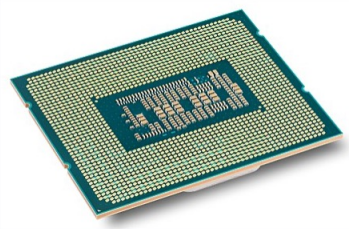
Rocket Lake:
Faster
Integrated Xe Graphics
Peripheral Component Interconnect Express (PCIe) interface 4.0

Alder Lake:
Hybrid cores
DDR5
PCIe 5.0

Rocket Lake

General information	
Launched	March 30, 2021; 23 months ago[2]
Discontinued	February 23, 2024; 11 months' time[1]
Product code	80708
Performance	
Max. CPU clock rate	1.3 GHz to 5.3 GHz
Cache	
L1 cache	80 KB per core: • 32 KB instructions • 48 KB data
L2 cache	512 KB per core
L3 cache	2 MB per core
Architecture and classification	
Architecture	x86-64
Microarchitecture	Cypress Cove
Instructions	x86, x86-64
Extensions	AES-NI, CLMUL, RDRAND, SHA, TXT MMX, SSE, SSE2, SSE3, SSSE3, SSE4, SSE4.1, SSE4.2 AVX, AVX2, AVX-512, FMA3 VT-x, VT-d
Physical specifications	
Transistors	Intel 14 nm FinFET process
Cores	Up to 8
Socket(s)	LGA 1200

Alder Lake



Intel Core i7-12700K

General information	
Launched	November 4, 2021; 16 months ago[1]
Marketed by	Intel
Designed by	Intel
Common manufacturer(s)	Intel
Product code	80715
Performance	
Max. CPU clock rate	1.0 GHz to 5.5 GHz, P-cores 700 MHz to 4.0 GHz, E-cores
Cache	
L1 cache	80 KB (32 instructions + 48 data), per P-core 96 KB (64 instructions + 32 data), per E-core
L2 cache	1.25 MB per P-core 2 MB per E-core module
L3 cache	Up to 30 MB, shared
Architecture and classification	
Technology node	Intel 7 (previously known as 10ESF)
Architecture	x86-64
Microarchitecture	Golden Cove (P-cores) Gracemont (E-cores)
Instruction set	x86-64

Imperial College London

Caches

- Caches near to the “cores” are small but close (and quick)
- The further away, the more expensive accessing memory is

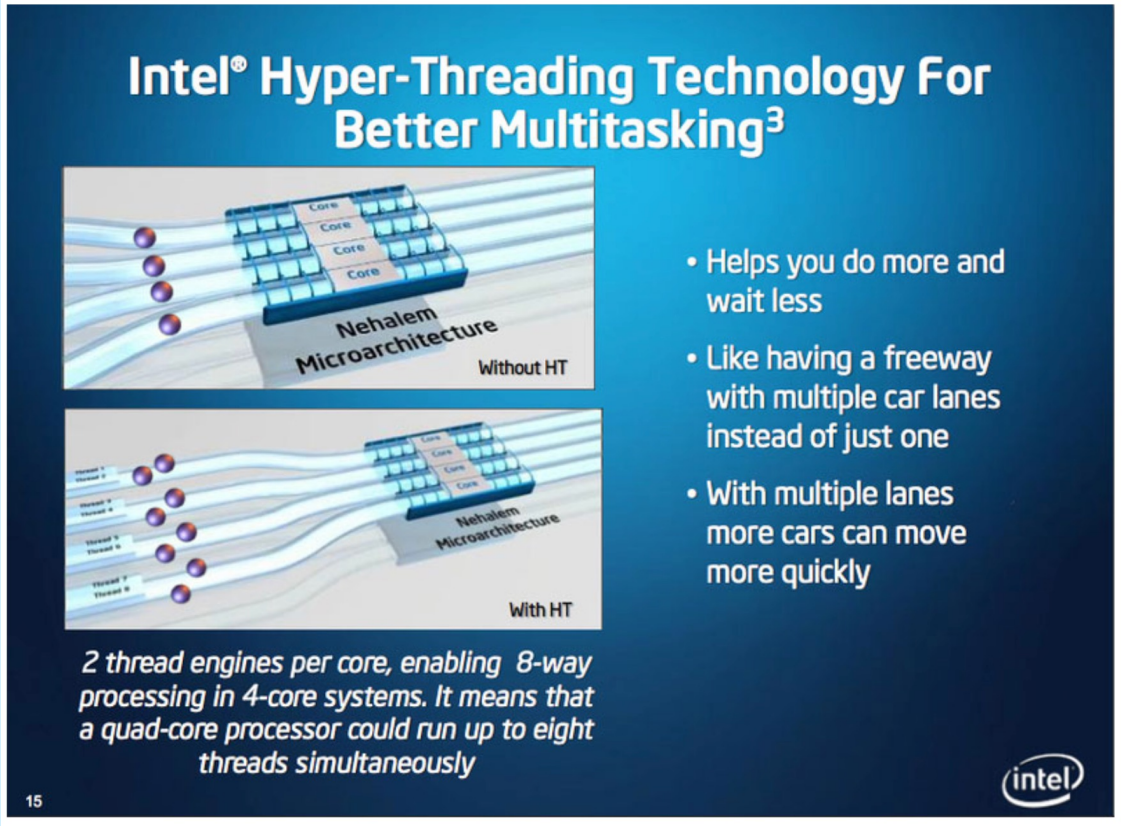
Core i7 Xeon 5500 Series Data Source Latency (approximate)

local L1 CACHE hit,	~4 cycles (2.1 - 1.2 ns)
local L2 CACHE hit,	~10 cycles (5.3 - 3.0 ns)
local L3 CACHE hit, line unshared	~40 cycles (21.4 - 12.0 ns)
local L3 CACHE hit, shared line in another core	~65 cycles (34.8 - 19.5 ns)
local L3 CACHE hit, modified in another core	~75 cycles (40.2 - 22.5 ns)
local DRAM	~60 ns

This is an example, access is faster now

Hyper-threading

- Hyper-Threading, developed by Intel, enables a single CPU core to execute multiple threads simultaneously, enhancing performance by leveraging parallelism.
- This technology optimises CPU utilisation, improving multitasking capabilities and system responsiveness in compatible hardware configurations.



Imperial College London

How can I exploit this?

Maximizing the use of the cache involves writing code that takes advantage of the principle locality, which refers to the fact that data and instructions that are accessed together tend to be located close to each other in memory.

```
const int ARRAY_SIZE = 1024 * 1024;
int data[ARRAY_SIZE];

// fill the data array with random values
for (int i = 0; i < ARRAY_SIZE; i++)
{ data[i] = rand(); }
// loop over the data array, summing up its values
int sum = 0; for (int i = 0; i < ARRAY_SIZE; i++)
{ sum += data[i]; }

cout << "The sum is: " << sum << endl;
```

If the start and end of the loop are defined, it is easier to optimize by the compiler

If all the data in my loop fits into L3 Cache, we are off to a good start

Imperial College London

Thoughts about the previous code

- In this code, the data array is relatively small at 1 million elements, which means it is likely to fit entirely in the L3 cache or even the L2 cache. This can help to maximize cache utilization and improve performance since the CPU can quickly access the necessary data from cache memory.
- However, if the data array were larger than the size of the L3 cache, some of the array's elements would be evicted from the cache to make room for other data. This could lead to cache thrashing, which can significantly degrade performance.
- Similarly, if the data array were much smaller than the size of the L1 cache, it is possible that the array would be entirely loaded into the L1 cache, making the loop over the array extremely fast since all the data would be immediately available in the L1 cache.
- In general, the size of the cache can have a significant impact on the performance of programs that rely heavily on data access. Therefore, optimizing code for cache usage requires careful consideration of the size and structure of the cache hierarchy in the target computer system.

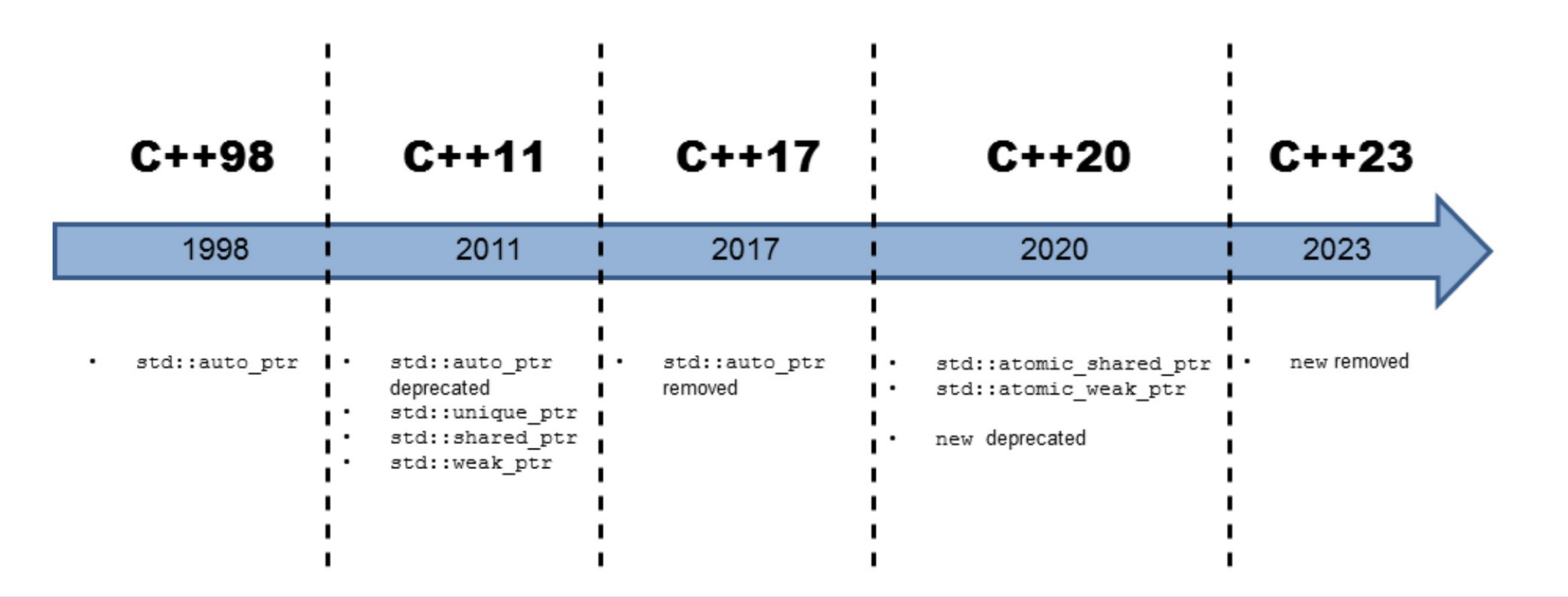
Imperial College London

Memory Management: Smart Pointers

- Now we will focus on memory management, and specifically learn about “smart pointers”
- Smart pointers can make memory management easier
- They keep track of how many other pointers point at their object
- And can automatically delete themselves when there no pointers left that reference their object
- No memory leaks!

Imperial College London

Evolution of pointers



Imperial College London

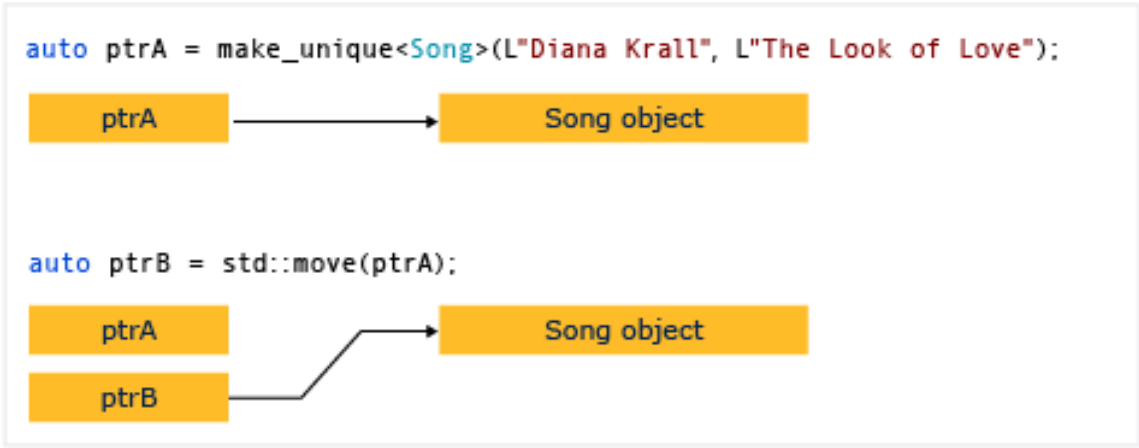
Smart Pointers

- We know about the problem of not calling delete on pointers that have been dynamically allocated with new
- Memory leaks!
- If your program runs for a long time or does a lot of things, this can be a big problem
- A few types of smart pointers
  - `unique_ptr<>`
  - `shared_ptr<>`
  - `weak_ptr<>`
- They are not a replacement for “raw” pointers
- You cannot do pointer arithmetic on them!

Imperial College London

`std::unique_ptr<>`

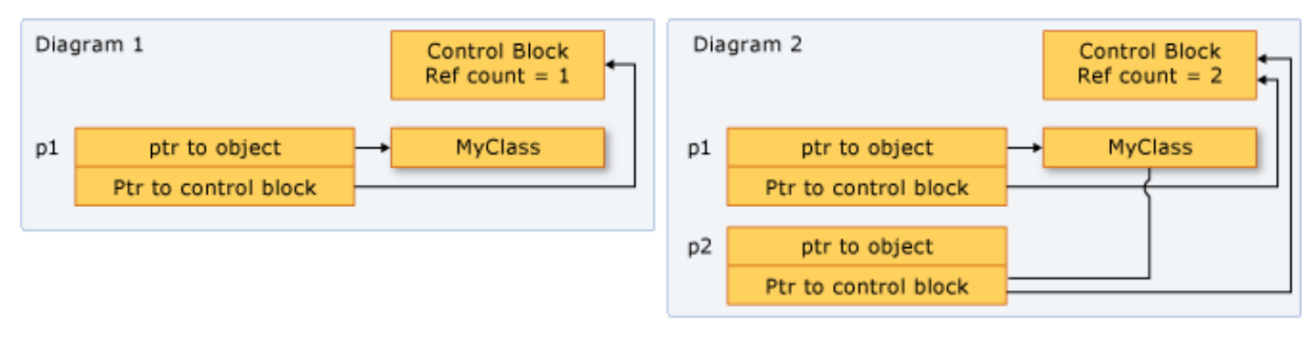
- A `std::unique_ptr<>` is by design as fast and as slim as a raw pointer but has a great benefit: it automatically manages its resource.
- `unique_ptr<>` can point at one thing only and deletes that object when it is deleted (e.g., if it goes out of scope)
- Makes sure there is only one copy of an object exists
- Does not support copying (….but you can move it!)



Imperial College London

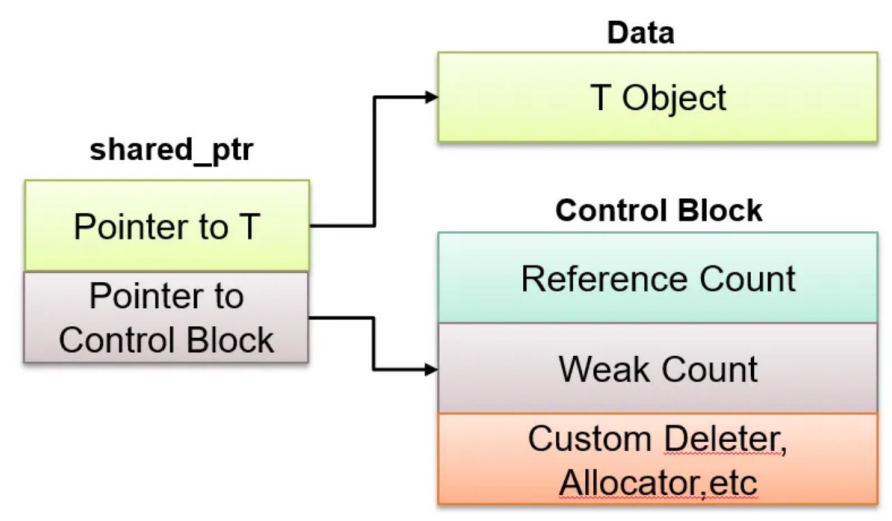
`shared_ptr<>`

- `shared_ptr<>` do “reference counting”. If you create a new `shared_ptr<>` pointing at the same object, it increments a counter
- When that counter equals 0, the destructor is called and the object destroyed
- The standard provides a macro to make allocating
- `shared_ptr` more efficient/convenient
- `make_shared<T>()` allocates memory for both our type and our reference counter close to each other in memory



Imperial College London

shared_ptr<>



The pointer is larger, it stores more information

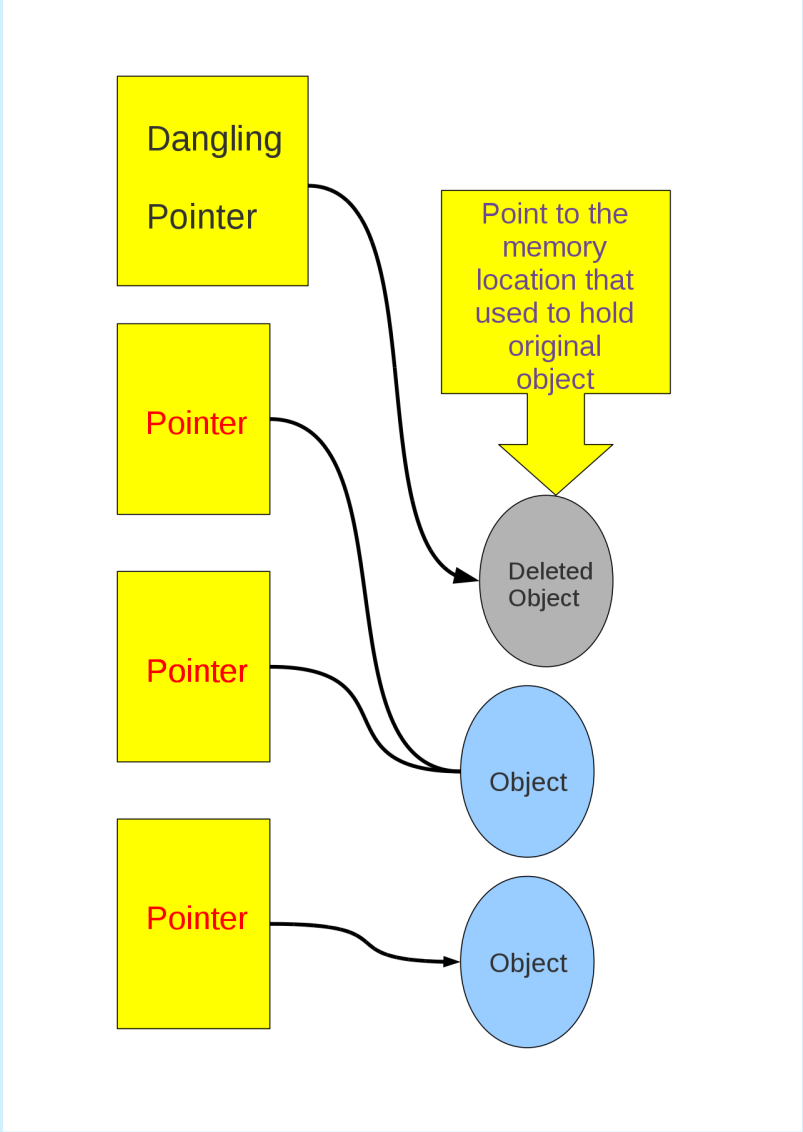
For later: Consider writing some code to compare performance between smart pointers and raw pointers?

Imperial College London

Consider this case

"Dangling pointers" point to regions in memory that have been deleted

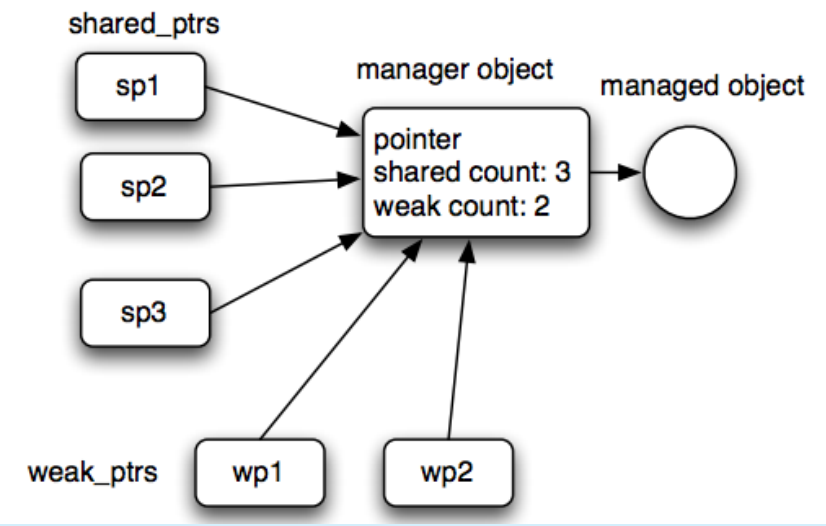
A std::shared_ptr can be used to manage data, and you can supplying std::weak_ptr to users of the data, the users can check validity of the data by calling expired() or lock().



Explanation from cppreference.com

weak_ptr<>

- std::weak_ptr is a smart pointer that holds a non-owning ("weak") reference to an object that is managed by std::shared_ptr. It must be converted to std::shared_ptr to access the referenced object.
- std::weak_ptr models temporary ownership: when an object needs to be accessed only if it exists, and it may be deleted at any time by someone else, std::weak_ptr is used to track the object, and it is converted to std::shared_pt to assume temporary ownership. If the original std::shared_pt is destroyed at this time, the object's lifetime is extended until the temporary std::shared_pt is destroyed as well.



Imperial College London

EXERCISE

- Consider the Matrix class in assessment_matrix.zip (Github)
- Read through the class code and understand what each function of the matrix class does
- Create a main that creates an instance of the matrix of dimensions 10 x 10. Fill the matrix with random values between 0 and 1. Print the matrix to screen.
- Can the vector values be a smart pointer? What type of smart pointer should it be (unique, shared or weak)? Why?

Imperial College London

Additional Slides

Imperial College London

Loop Unrolling

Article Full-text available

Compiling for Reconfigurable Computing: A Survey

June 2010 · ACM Computing Surveys 42(4)
DOI: 10.1145/1749603.1749604
Source · DBLP

João M P Cardoso · Pedro Diniz · Markus Weinhardt

```
#define W 3
...
int a[W][W], b[W][W], c[W];
...
for(x=0; x < W; x++) {
      sum = 0;
      for(y=0; y < W; y++) {
            sum += (a[x][y] * b[y][x]);
      }
      c[x] = sum;
}
```

(a)

```
#define W 3
...
int a[W][W], b[W][W], c[W];
...
for(x=0; x < W; x++) {
      sum  = (a[x][0] * b[0][x]);
      sum += (a[x][1] * b[1][x]);
      sum += (a[x][2] * b[2][x]);
      c[x] = sum;
}
```

(b)

Figure

Caption

Fig. 6. Loop unrolling and opportunities for parallel execution: (a) original C source code; (b) full unrolling of inner loop. —Code motion across conditionals: This is the most used type of code motion. Recent techniques have been applied in the CDFG and HTG for HLS of ASICs [Gupta et al. 2001; Santos et al. 2000], considering constraints ... Read more

Imperial College London

Loop Unrolling 2

Article | Full-text available

Compiling for Reconfigurable Computing: A Survey

June 2010 · ACM Computing Surveys 42(4)
DOI: 10.1145/1749603.1749604
Source · DBLP

João M P Cardoso · Pedro Diniz · Markus Weinhardt

```
...
int img[N][N];
...
for(j=0; j < N; j++) {
    ...
    for(i=0; i < N; i++) {
        ... = img[j][i];
    }
    ...
}
```
(a)

```
...
int imgOdd[N][N/2], imgEven[N][N/2];
...
for(j=0; j < N; j++) {
    ...
    for(i=0; i < N; i+=2) {
        ... = imgOdd[j][i/2];
        ... = imgEven[j][i/2];
    }
    ...
}
```
(b)

Figure

Caption

Fig. 7. Loop unrolling and array data distribution example: (a) original C source code; (b) loop unrolled by 2 and distribution of img.

Imperial College London

Loop Unrolling 3

Article | Full-text available

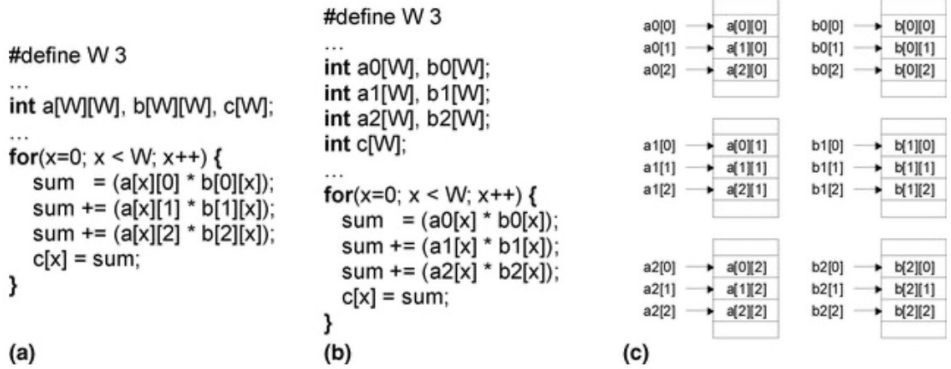
Compiling for Reconfigurable Computing: A Survey

June 2010 · ACM Computing Surveys 42(4)

DOI: 10.1145/1749603.1749604

Source · DBLP

João M P Cardoso · Pedro Diniz · Markus Weinhardt



Figure

Caption

Fig. 17. Bank disambiguation as a form to increase the memory bandwidth and the ILP degree when targeting architectures that support various memory banks accessed at the same time: (a) original C code; (b) C code after unrolling and bank disambiguation; (c) arrays mapped to six memories and the correspondence of each arr ... Read more

Why is it important?

- Again, why do we care? Caches are completely transparent to the user, so how can we benefit from thinking about them?
- When you access memory, the CPU automatically grabs neighbouring memory and stores it in the cache
- If you try to order your loops (etc) in your algorithm to use data close in memory, you will be rewarded!

Imperial College London

Coding Strategies for Cache Performance

- Cache locality: Keep frequently used data in memory locations that are close to each other.
- Avoid pointer chasing: Avoid having to traverse multiple pointers to access a piece of data. This can cause cache thrashing (bad for performance). Use data structures with direct access to data.
- Minimize data movement: Avoid moving data around unnecessarily. This can cause cache invalidations.
- Use data structures that are cache-friendly: For example, a linear search through a linked list can be slow due to poor cache locality, while a binary search through an array can be faster due to better cache performance.
- Use compiler optimizations: Modern compilers can perform optimizations that can help improve cache performance. For example, the compiler can automatically vectorize code or reorder operations to maximize cache locality.

Imperial College London

EXERCISE

- Create an array D of size `s`=1000 `double`s. Through the console, request a number `n` from 1 to 10. Once read, resize the array so that its new size is `s*n`. Fill the new entries of the array with random numbers between 0 and 1. What is the size of array D? How large can it become?

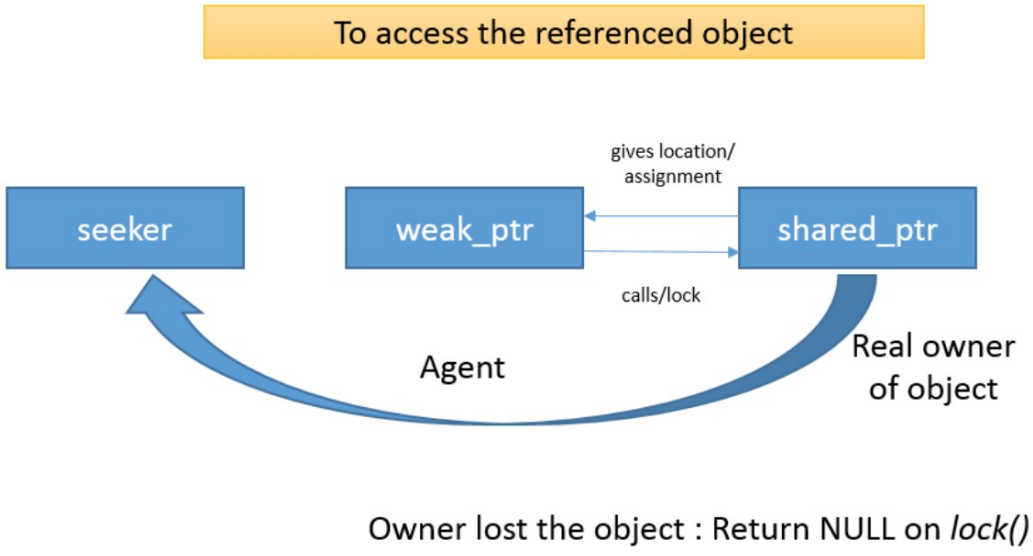
Imperial College London

More on `weak_ptr<>`

- `weak_ptr<>` is like a `shared_ptr<>`, but without ownership
- You can give someone a `weak_ptr<>`, and regardless of how many you give out, it doesn't change your "strong" reference counter
- You still control ownership of your memory
- A "Get" method can be defined so that it returns a `shared_ptr<>` to our values array, when we tried to delete values other people may have a `shared_ptr<>` to it
- The reference may not be zero and it might not be deleted when we want

Imperial College London

Weak pointer – practical example



Roughly speaking, `weak_ptr` role is similar to the role of *housing agency*. Without agents, to get a house on rent we may have to check random houses in the city. The agents make sure that we visit only those houses which are *still accessible and available* for rent.

https://stackoverflow.com/questions/12030650/when-is-stdweak-ptr-useful

answered Oct 21, 2016 at 14:02
Saurav Sahu
12.6k • 5 • 59 • 78

Imperial College London

`weak_ptr<>`

- You have no idea if the thing a `weak_ptr<>` is pointing to is going to be destroyed halfway through you working on it though
- Can call `expired()` to test if the reference count is zero
- But again, you have no guarantee the `weak_ptr<>` stays valid after you've called `expired()`
- Calling `lock()` returns a `shared_ptr<>` if the reference count is non-zero, a `nullptr` if nots
- Normally if you want to work with a `weak_ptr`, you normally "promote" it to a `shared_ptr<>`, then make sure you delete a `shared_ptr` when you are done

Imperial College London

Memory Management: Recap

- When you “create” a variable there are two places it could be stored: The stack and the heap
- All memory allocated on the stack is known at compile time
- The heap stores things that aren’t known at compile time