
UNSW
SYDNEY
Australia's
Global
University
COMP1911 - Computing 1A

9. Pointers and Memory

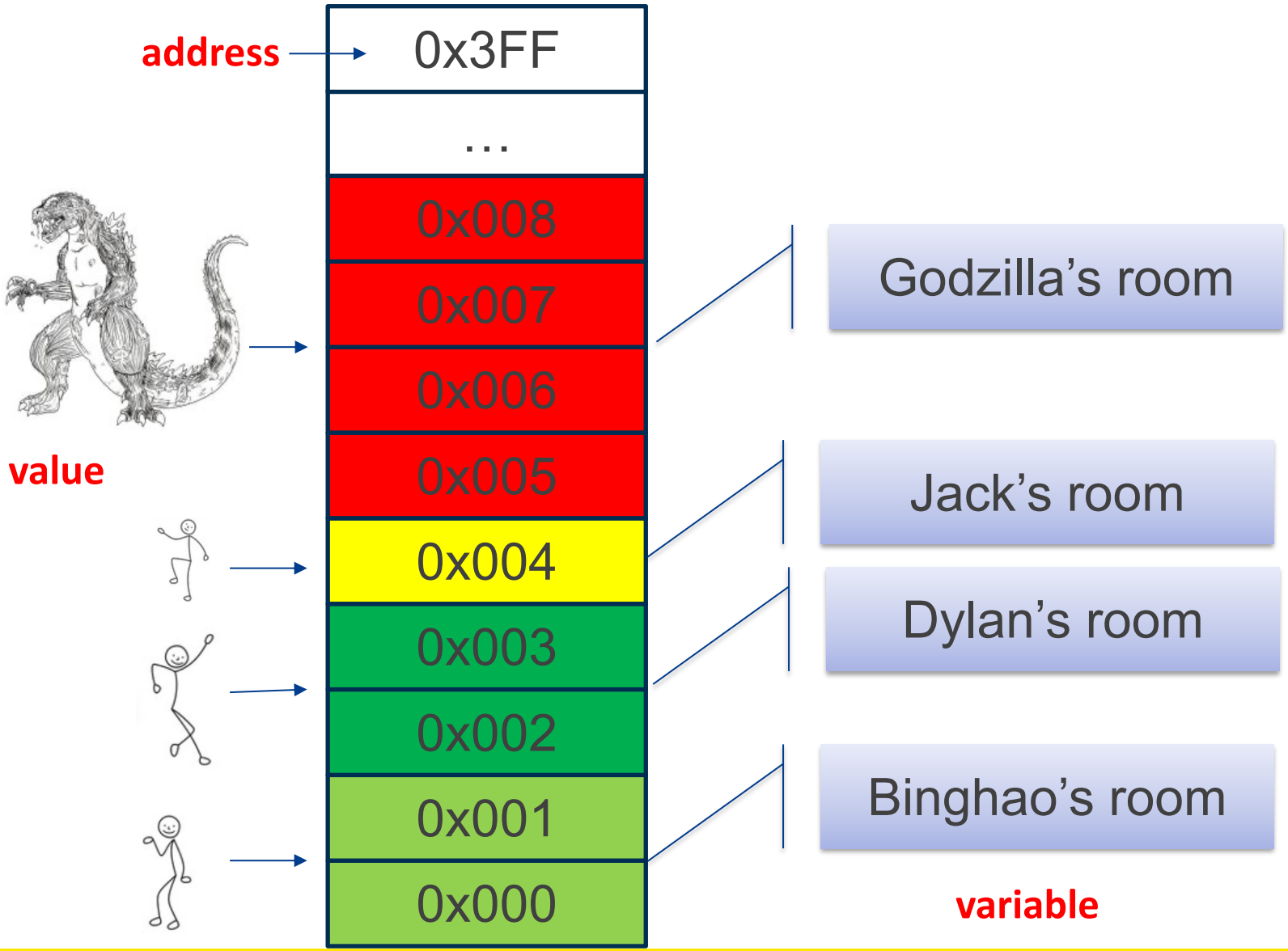
What is a pointer

- For some people, pointers in C are easy and fun to learn.
- For some people, pointers are genuinely hard.
- Pointer is important. C is a high level language, but can deal with the memories like the assembly language
  - Some C programming tasks are performed more easily with pointers.
  - Some tasks, such as dynamic memory allocation, cannot be performed without using pointers.

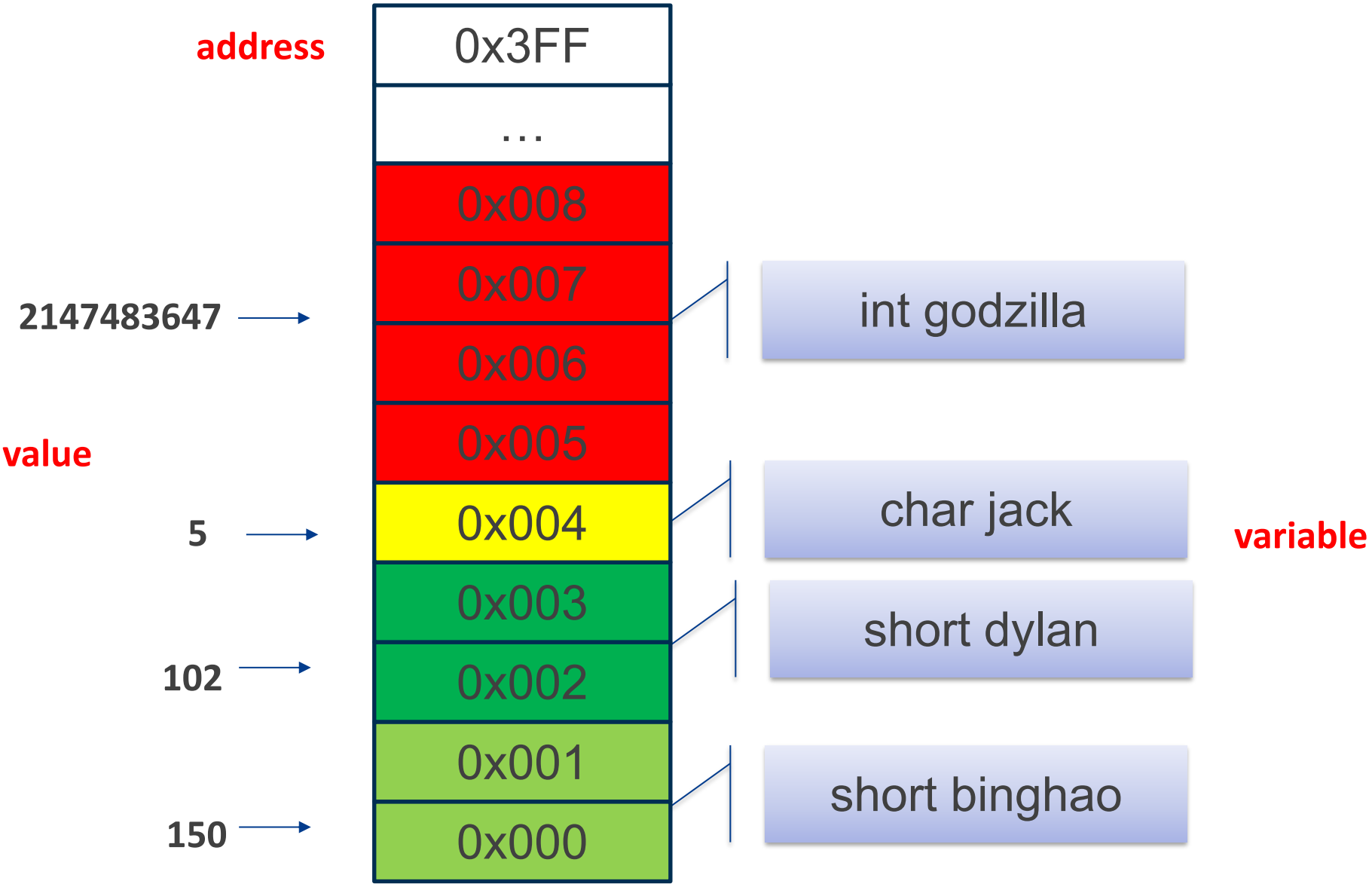
What is a **pointer**?

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the **memory** location.

UNSW Princes

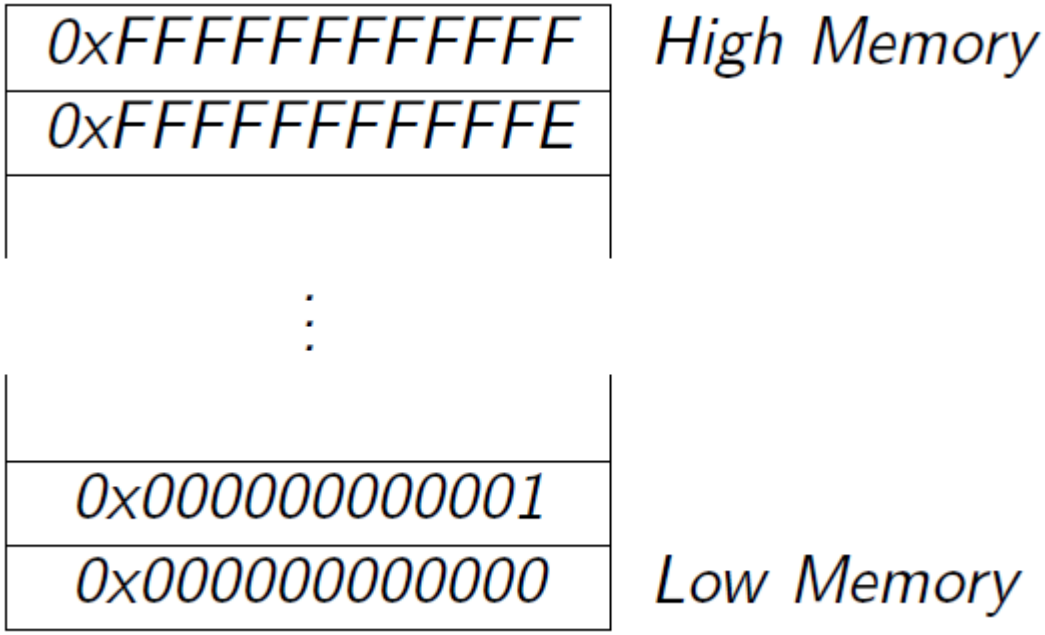


Computer memory



Memory Organisation

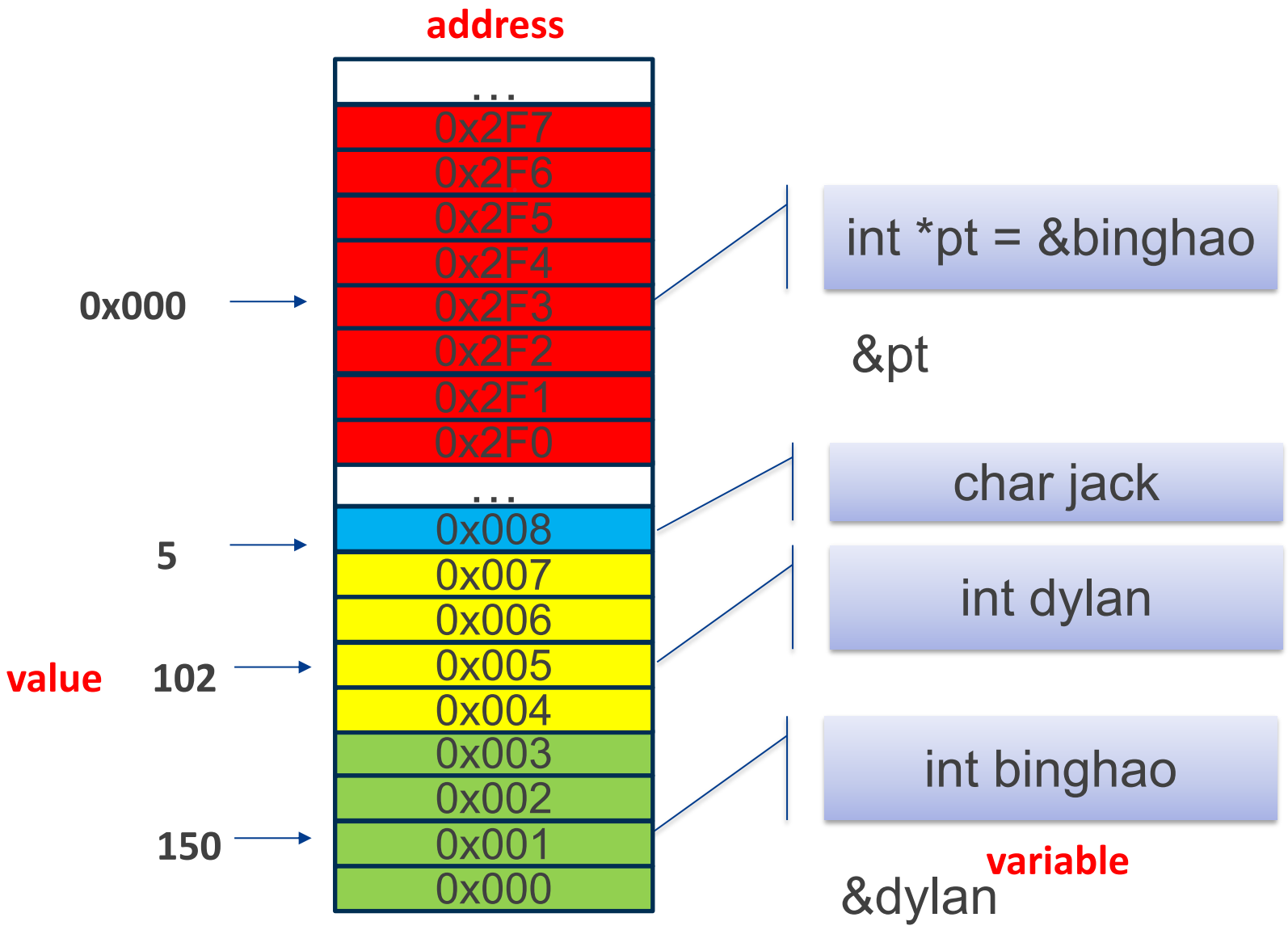
In order to fully understand how pointers are used to reference data in memory, here's a few basics on memory organisation.



Memory

- Computer memory is a large array of consecutive data cells or bytes
- A variable will stored in 1 or more bytes
- On CSE machines an *int* occupies 4 bytes, a *double* 8 bytes
- The & (address-of) operator returns a reference to a variable.
- Almost all C implementations & operator returns a reference
- It is convenient to print memory addresses in Hexadecimal notation

Variable address and pointer



Variables in Memory

```
int k,m;

printf( "address of k is %p\n", &k );
printf( "address of m is %p\n", &m );
```

Output:

```
address of k is 0xbfffffffb80
address of m is 0xbfffffffb84
```

This means that

- k occupies the four bytes from 0xbfffffffb80 to 0xbfffffffb83
- m occupies the four bytes from 0xbfffffffb84 to 0xbfffffffb87

%p can be simply understood as format specifier to print the hexadecimal memory location (address) of a variable.

Arrays in Memory

Elements of the array will be stored in consecutive memory locations:

```
int a[5];

i=0;
while (i < 5) {
   printf("address of a[%d] is %p\n", i, &a[i]);
}
```

```
address of a[0] is 0xbfffffffb60
address of a[1] is 0xbfffffffb64
address of a[2] is 0xbfffffffb68
address of a[3] is 0xbfffffffb6c
address of a[4] is 0xbfffffffb70
```

What is the value of a?

Pointers

A pointer is a data type whose value is a reference to another variable.

```
int *ip;    // pointer to int
char *cp;   // pointer to char (more next week)
double *fp; // pointer to double
```

In most C implementations, pointers store the memory address of the variable they refer to.

How many bytes are needed to store the int pointer, double pointer and char pointer?

Total Results: 0

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Size of a Pointer

Just like any other variable of a certain type, a variable that is a pointer also occupies space in memory. The number of memory cells needed depends on the computer's architecture. For example:

- 32-bit platform pointers likely to be 4 bytes
  e.g. CSE lab machines
- 64-bit platform pointers likely to be 8 bytes
  e.g. many student machines
- tiny embedded CPU pointers could be 2 bytes
  e.g. your microwave

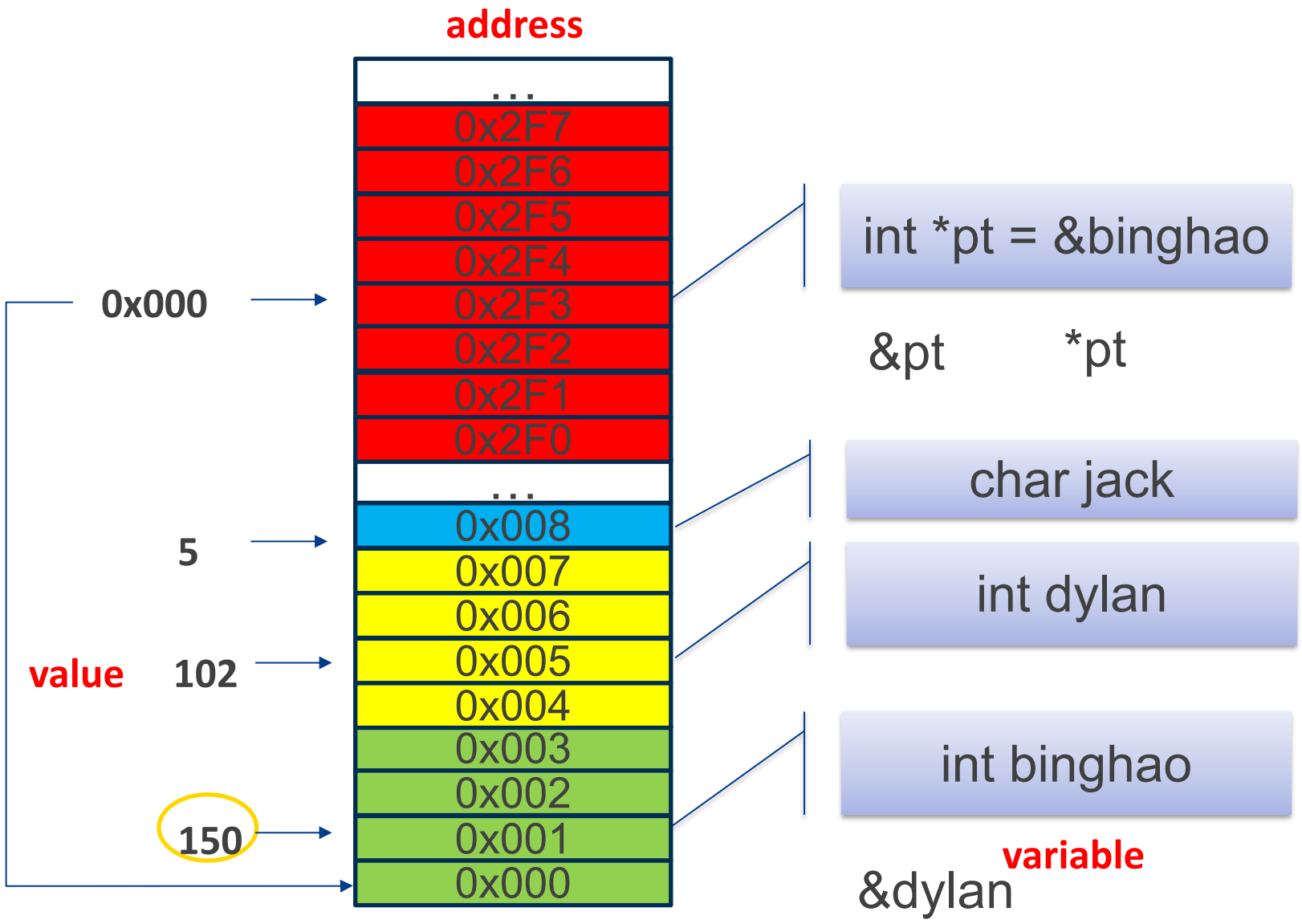
Pointers

The & (address-of) operator returns a reference to a variable.
The * (dereference) operator accesses the variable referred to by the pointer.

For example:

```
int i = 7;
int *ip = &i;
printf("%d\n", *ip); // prints 7
```

Variable address and pointer



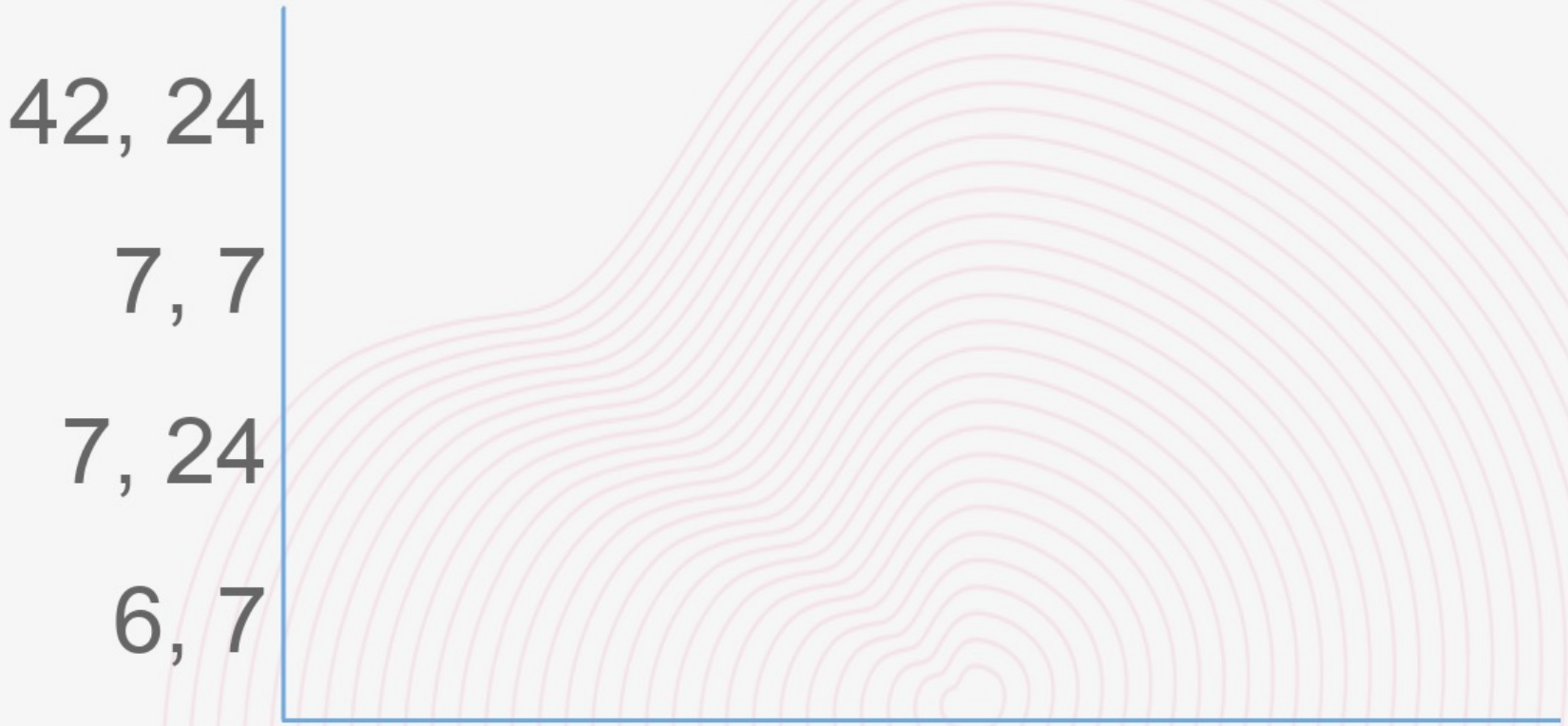
Pointers

The & (address-of) operator returns a reference to a variable.
The * (dereference) operator accesses the variable referred to by the pointer.

For example:

```
int i = 7;
int *ip = &i;
printf("%d\n", *ip); // prints 7
*ip = *ip * 6;
printf("%d\n", i);
i = 24;
printf("%d\n", *ip);
```

What will be printed out?



Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Pointers

The & (address-of) operator returns a reference to a variable.
The * (dereference) operator accesses the variable referred to by the pointer.

For example:

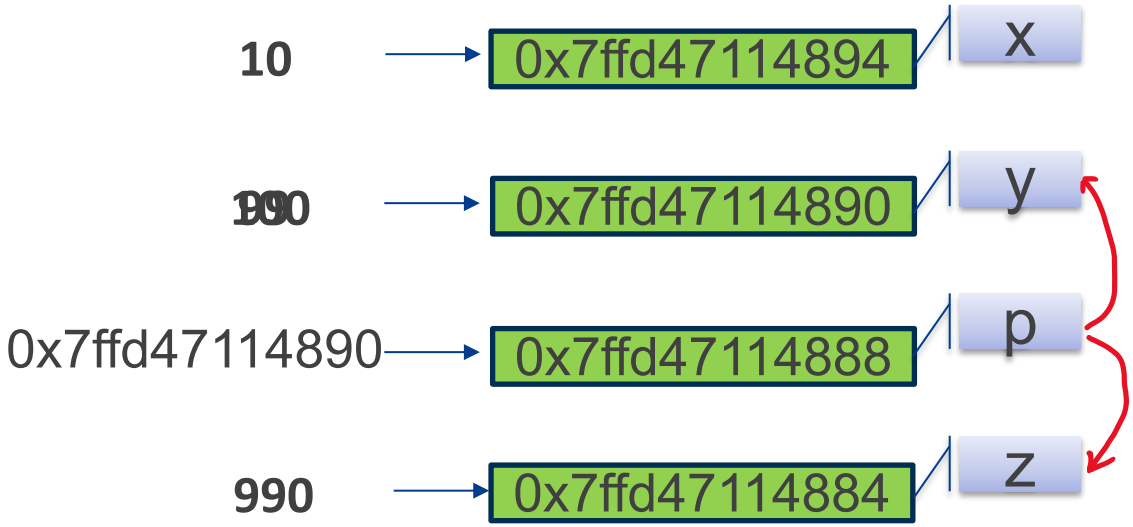
```
int i = 7;
int *ip = &i;
printf("%d\n", *ip); // prints 7
*ip = *ip * 6;
printf("%d\n", i);   //prints 42
i = 24;
printf("%d\n", *ip); // prints 24
```

Pointers

- Like other variables, pointers need to be initialised before they are used .
- Like other variables, its best if novice programmers initialise pointers as soon as they are declared.
- The value NULL can be assigned to a pointer to indicate it does not refer to anything.
- NULL is a #define in stdio.h
- NULL and 0 interchangeable (in modern C).
- Most programmers prefer NULL for readability.

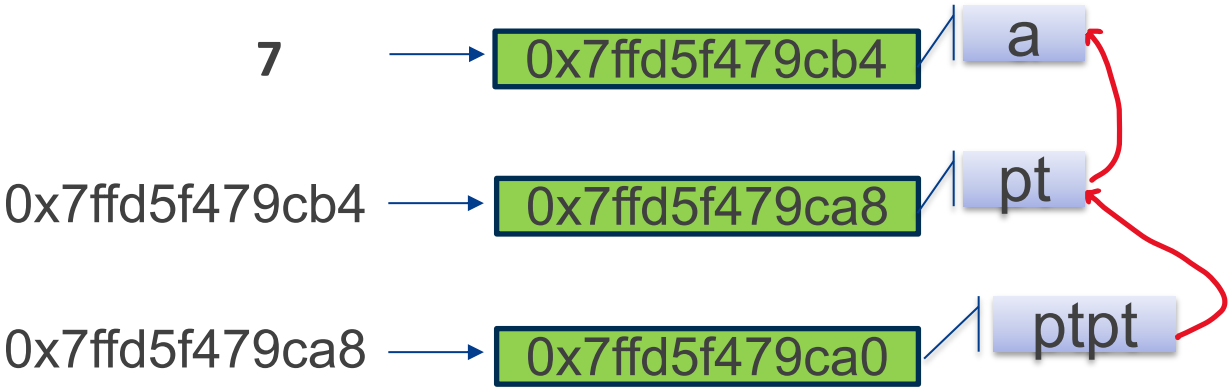
Pointers

```
int x = 10;
int y = 99;
int *p = &y;

int z = *p * x;

*p = *p + 1;

p = &z;

printf("%d %d %d %d\n",x,y,z,*p);
```



Pointers

```
int a = 7;
int *pt = &a;
int **ptpt = &pt;

printf("%lx \n", *ptpt);
printf("%p \n", pt);
printf("%d \n", **ptpt);
```



Pointer Arguments

We've seen that when primitive types are passed as arguments to functions, they are passed by value and any changes made to them are not reflected in the caller.

```
void increment(int n) {
  n = n + 1;
}
```

This attempt fails. But how does a function like scanf manage to update variables found in the caller? scanf takes pointers to those variables as arguments!

```
scanf("%d", &answer);
```

increment (&n);

```
void increment(int *n) {
  *n = *n + 1;
}
```

Pointer Arguments

Passing by reference

We use pointers to pass variables by *reference*! By passing the address rather than the value of a variable we can then change the value and have the change reflected in the caller.

```
int i = 1;
increment(&i);
printf("%d\n", i); //prints 2
```

In a sense, pointer arguments allow a function to 'return' more than one value. This greatly increases the versatility of functions. Take scanf for example, it is able to read multiple values and it uses its return value as error status.

Pointer Arguments

Classic Example

Write a function that swaps the values of its two integer arguments.

Before we knew about pointer arguments this would have been impossible, but now it is straightforward.

```
void swap(int *n, int *m) {
  int tmp;

  tmp = *n;
  *n = *m;
  *m = tmp;
}
```

Pointer Return Value

- You should not find it surprising that functions can return pointers. However, you have to be extremely careful when returning pointers. Returning a pointer to a local variable is illegal - that variable is destroyed when the function returns.

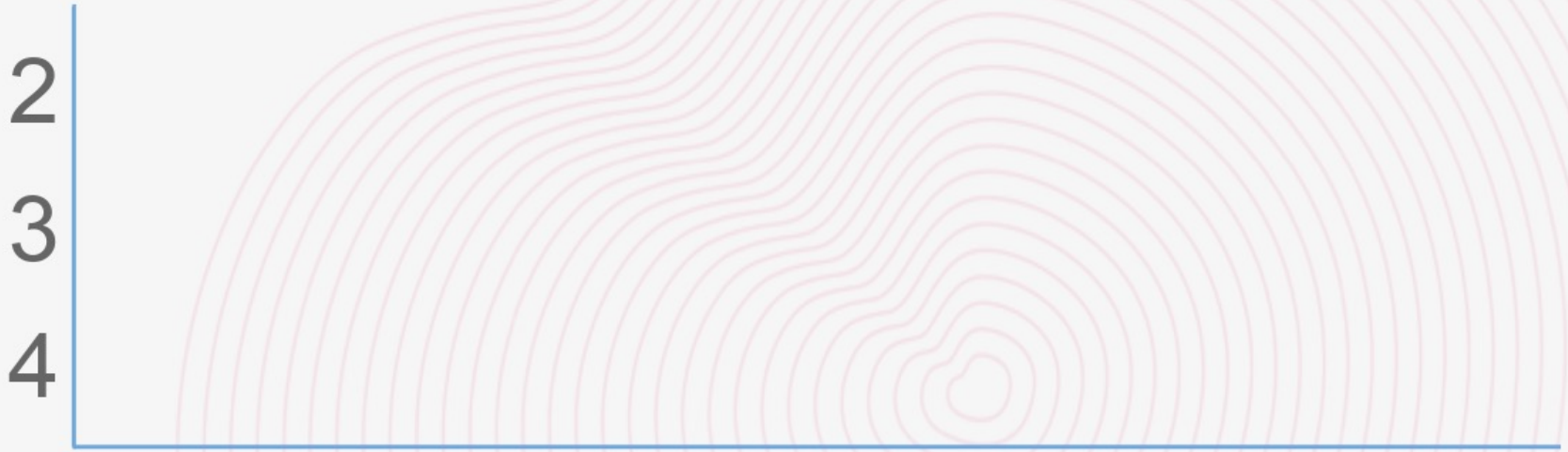
- But you can return a pointer that was given as an argument:

```
int * increment(int *n) {
  *n = *n + 1;
  return n;
}
```

- Nested calling is now possible: increment(increment(&i));

What is the output?


```
int i=2;
increment(increment(&i));
printf("i is %d", i);
```



Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Array Representation

A C array has a very simple underlying representation, it is stored in a contiguous (unbroken) memory block and a pointer is kept to the beginning of the block.

```
int s[] = {1,2,3};
printf("s:\t%p\t*s:\t%d\n\n", s, *s);
printf("&s[0]:\t%p\ts[0]:\t%d\n", &s[0], s[0]);
printf("&s[1]:\t%p\ts[1]:\t%d\n", &s[1], s[1]);
printf("&s[2]:\t%p\ts[2]:\t%d\n", &s[2], s[2]);
// prints
// s: 0x7fff4b741060 *s: 1
// &s[0]: 0x7fff4b741060 s[0]: 1
// &s[1]: 0x7fff4b741061 s[1]: 2
// &s[2]: 0x7fff4b741062 s[2]: 3
```

Array variables act as pointers to the beginning of the arrays!

Note: The '\t' is called escape sequence(a character) which represent the "tab" character of your keyboard.

Array Representation

Since array variables are pointers, it now should become clear why we pass arrays to scanf without the need for address-of (&) and why arrays are passed to functions by reference!
We can even use another pointer to act as the array name!

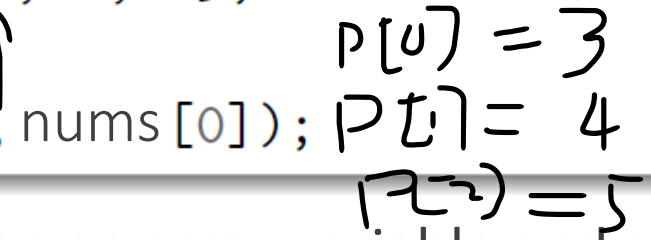
```
int nums[] = {1, 2, 3, 4, 5};
int *p = nums;

printf("%d\n", nums[2]);
printf("%d\n", p[2]);
//both print 3
```

Since nums acts as a pointer we can directly assign its value to the pointer p

Array Representation

We can even make a pointer point to the middle of an array:

```
int nums[] = {1, 2, 3, 4, 5};
int *p = &nums[2];
printf("%d %d\n", *p, nums[0]);
```



So is there a difference between an array variable and a pointer?

```
int i = 5;
p = &i;     // this is OK
nums = &i; // this is an error
```

Unlike a regular pointer, an array variable is defined to point to the beginning of the array, it is constant and may not be modified.

Pointer Comparison

Pointers can be tested for equality or relative order.

```
double ff[] = {1.1, 1.2, 1.3, 1.4, 1.5, 1.6};
double *fp1 = ff;
double *fp2 = &ff[0];
double *fp3 = &ff[4];

printf("%d %d\n", (fp1 > fp3), (fp1 == fp2));
```

What are printed?

Total Results: 0

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Pointer Comparison

Pointers can be tested for equality or relative order.

```
double ff[] = {1.1, 1.2, 1.3, 1.4, 1.5, 1.6};
double *fp1 = ff;
double *fp2 = &ff[0];
double *fp3 = &ff[4];

printf("%d %d\n", (fp1 > fp3), (fp1 == fp2));
//prints 0 1
```

Note that we are comparing the values of the pointers, i.e., memory addresses, not the values the pointers are pointing to!

Beware: Don't Try to Return an Array

It might be tempting to try returning an array from a function:

```
int[] foo(void) {
        int nums[] = {1,2,3};
        return nums;
}
```

This looks good but fails spectacularly!

Careful

Arrays are passed by reference, so a pointer to the local array nums is returned. The array is destroyed immediately after the return statement. Using it in the caller then becomes a run-time error!
It is possible to return dynamically allocated arrays, which we will learn later in the course.

Pointer Summary

Pointers:

- are a compound type
- usually implemented with memory addresses
- are manipulated using address-of(&) and dereference(*)
- should be initialised when declared
- can be initialised to NULL
- should not be dereferenced if invalid
- are used to pass arguments by reference
- are used to represent arrays
- should not be returned from functions if they point to local variables

Memory Organisation

- During execution programs variables are stored in memory.
- Memory is effectively a gigantic array of bytes.

  COMP1521 will explain more
- Memory addresses are effectively an index to this array of bytes.
- These indexes can be very large

  up to 2^{32} - 1 on a 32-bit platform

  up to 2^{64} - 1 on a 64-bit platform
- Memory addresses usually printed in hexadecimal (base-16).

Questions