Arrays, Pointers, and Dynamic Memory Allocation in C++

CPSC 221 Supplementary Notes

See also our textbook:

Koffman, Elliot B. and Wolfgang, Paul A.T. *Objects, Abstraction, Data Structures and Design Using C++*. Wiley, 2006.

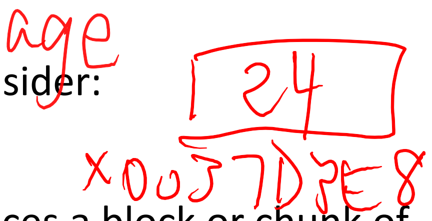
Annotated Slides
from Ed's class

Addresses and Pointers

- When implementing data structures in C++, we often need to consider *addresses* and *pointers*.
- You can think of addresses in memory in a way analogous to rooms in a hotel, or houses in a city. Each room or house has a specific location that is identified by its room number or street address. For example, if we want to put something in a specific house, then we need to know *which* house. The address tells us which house it is.
- Similarly, each storage location in memory (RAM) has an address associated with it. You can view RAM as one huge array of memory cells. Each cell has an address (e.g., at the byte level). The address is the location in memory where a given variable or identifier stores its data.
  - In the case of an array, a number of consecutive storage locations of a given data type is reserved as a bigger block.

Addresses and Pointers (cont.)

- Each byte of memory has a unique *address*. Consider:



- `int age = 24;`
- age is an identifier (variable name) that references a block or chunk of memory that is used to store the integer *value* 24. Rather than referencing this block using a hard-to-remember numeric *address* like x0037D3E8, we use the identifier age instead; this makes programming much easier.
- At *compile time*, the compiler knows how much memory to allocate to age (i.e., 4 bytes for an integer). Four bytes can hold integers in the range of approximately +/− 2 billion. Why?
  - 4 bytes = 32 bits
  - The first bit is the sign; the other 31 bits allow for 2^{31} choices $\approx$ 2 billion
- If an address is 4 bytes long, how many bytes of RAM can it address? Many modern machines/compilers use 8 bytes for addresses (e.g., 64-bit machines).

32 bit addresses

1 bit => 1 byte of memory

#define MAX 1000
...
double myArray[MAX];

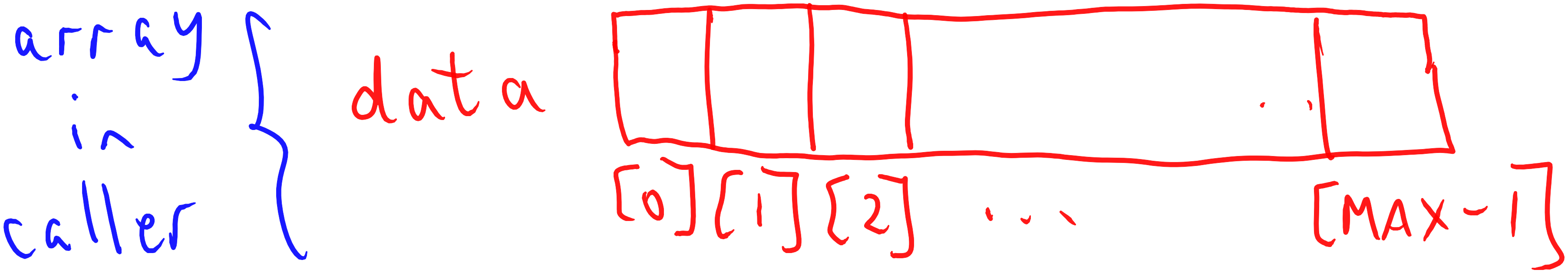
Addresses and Arrays

- Arrays are always assumed to be pass-by-reference. For example:

```
• double getMaximum(double data[], int size);  // prototype
  double getMaximum(double * data, int size);  // equiv. prototype
• answer = getMaximum(myArray, length);        // function call
```

e.g., 15

- We're passing the address of the start of the array to the function. In other words, we're passing a pointer to the array's first element. ✓
- Suppose the array were set up to hold 100,000 items. You can view this as a linear array of memory cells starting with cell 0 (i.e., subscript (or index) 0 to yield `data[0]`), and ending with cell 99,999 (i.e., subscript 99999 to yield `data[99999]`).
- Cell 0 is the starting address of the array.



Addresses and Arrays (cont.)

- Let’s summarize the advantages for the case on the previous page.

```
double getMaximum(double data[], int size);  // prototype
answer = getMaximum(myArray, length);        // function call
```

- Suppose the array were set up to hold 100,000 items. Here are some advantages to note, using call-by-reference compared to call-by-value:
  - First, the programmer doesn’t have to code 100,000 individual variable names or locations (like `data[0], data[1], ..., data[99999]`).

    time too !
  - This can save a lot of memory space in the function/program; otherwise, some programs would be massive.
  - It also avoids needless copying at execution time, especially since the function’s local data will be destroyed upon return, anyway … and be re-copied/re-initialized every time it was called!
    - Imagine the function intentionally being called millions of times in a loop.
- Why isn’t every variable passed by reference?

Pointer Examples

- A pointer is a data type that contains the address of the object in memory, but it is *not* the object itself. We declare a pointer to an object in the following way:

```
dataType *identifier;
```

- For example, to declare a pointer to an integer, we can do the following:

```
int *intPtr;  or  int* intPtr;  or  int * intPtr;
```

  - Note that the location of the * is somewhat flexible. (Whitespace is ignored in C.)

- **Warning:** The declaration:

```
int *var1, var2;
```

… declares var1 to be a *pointer* to an integer, but var2 to be an *integer*! To declare both as pointers, do the following, or just do one per line:

```
     int  *var1, *var2;
or   int * var1;
     int * var2;
```

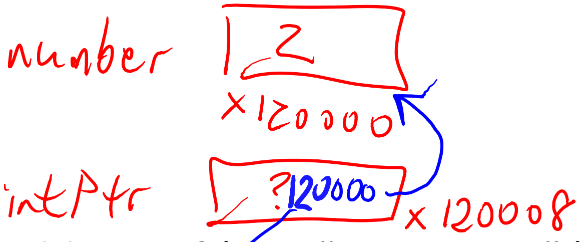
int var2;

Pointer Examples (cont.)

Example 1:

- Consider the following code segment:

```
int     number = 2;
int   * intPtr;
```



- How do we get `intPtr` to contain the address of (i.e., “to point to”) `number`? This is achieved using the `&` (address-of) operator:

```
intPtr = &number;
```

$\underbrace{\text{sizeof(int *)}}_{8}$

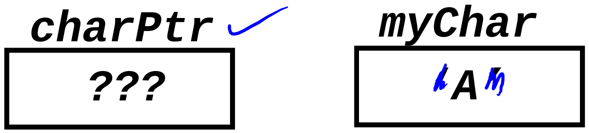
- Now that `intPtr` is pointing to `number`, how do we reference the object pointed to by `intPtr`? This is achieved using the `*` (dereferencing) operator:

```
cout << *intPtr << endl;
```

$* 120000 = \underline{\underline{2}}$

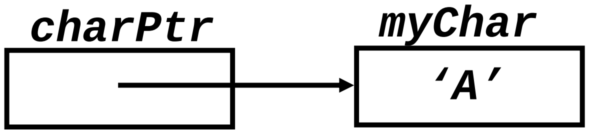
Example 2:

This time, we'll point to a character data type:



```
char * charPtr;
char   myChar = 'A';
```

Note that no address is contained in charPtr yet!

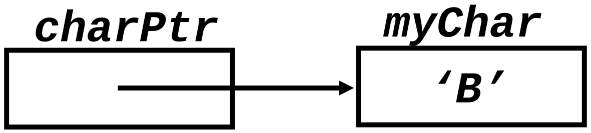


```
charPtr = &myChar;
```

We can change the value of myChar directly (in the usual way) … or indirectly using the * or "dereferencing" operator:

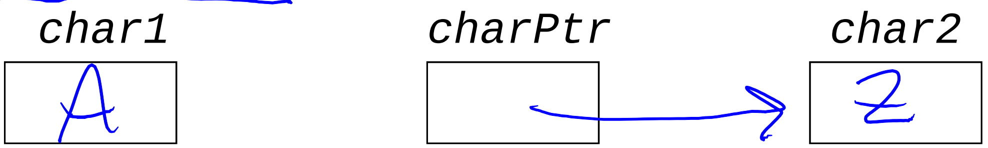
myChar ='B';

```
*charPtr = 'B';
```

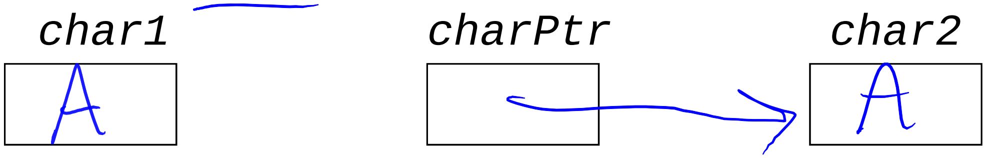


Exercise: Fill in the boxes/arrows.

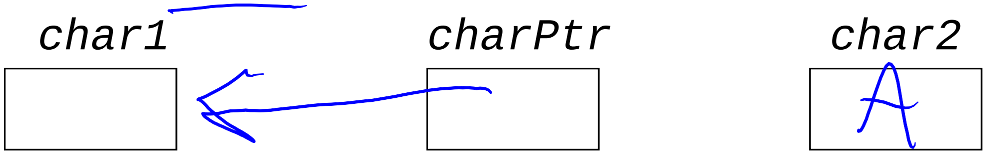
```
char    char1 = ‘A’, char2 = ‘Z’;
char *  charPtr = &char2;
```



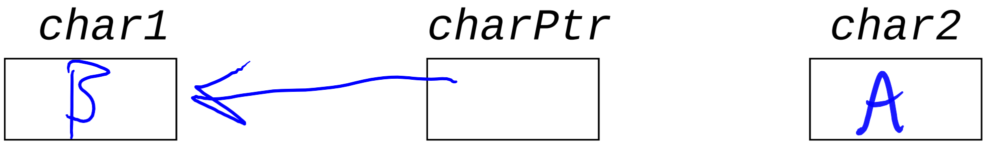
```
*charPtr = char1;
```



```
charPtr = &char1;
```



```
*charPtr = ‘B’;
```



Simplifications

- To simplify things, we are taking some liberties in our discussion on the following pages.
- A few notes:
  - Field lengths and address alignments (starting points) are machine and compiler dependent. For example, addresses on a 32-bit machine will be 4 bytes long; but on a 64-bit machine, they'll be 8 bytes long.
  - Addresses for variables may be in descending, rather than ascending, order (because of how they're allocated on the memory stack).
  - Data types may be *aligned* on different boundaries or relative addresses than shown.
  - If you re-run a program, you'll often get different addresses/offsets.
    - This may be done for security reasons.

Memory Consumption

- Let us assume that addresses (or pointer data types) are always 8 bytes long on our machine. Here is an example that shows addressing in relative terms:

Data Type/Example	Description	Size
int	integer	4 bytes
double	double precision floating point	8 bytes
char	single character	1 byte
float	single precision floating point	4 bytes
int *	pointer to another memory location that holds an integer	8 bytes
double *	pointer to … a double	8 bytes
char *	pointer to … a character	8 bytes
char x[10];	x is an array of 10 characters	10 bytes
int age[30];	age is an array of 30 integers	30 * 4 bytes = 120 bytes

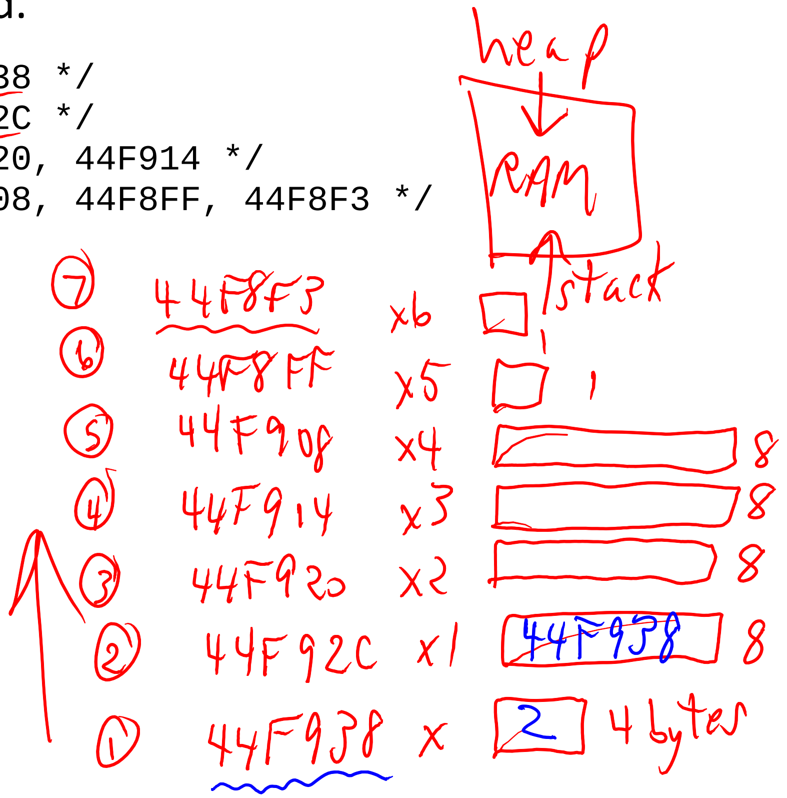
Example: Determine the Output

- Suppose we have the following declarations and addresses. Draw a picture of memory, and determine the output that's produced.

```
int     x;              /* memory location 44F938 */
int     *x1;            /* memory location 44F92C */
int     *x2, *x3;       /* memory location 44F920, 44F914 */
char    *x4, x5, x6;    /* memory location 44F908, 44F8FF, 44F8F3 */
...
x = 2;
x1 = &x;
cout <<   x << endl;
cout <<  &x << endl;
cout <<  x1 << endl;
cout << *x1 << endl;
// Use printf or specify (void *) to avoid
// a printing problem with cout for char's.
cout << (void *) &x6 << endl;
```

- Output:

2
44F938
44F938
2
44F8F3



Dynamic Memory Allocation

- Problem:
  - Sometimes we don't know *how much* memory we need at compile time.
  - Every user may use the program differently. For example, suppose you allocate an array capable of holding 1000 integers, and you hardcode the value "1000".
    - What if the user plans to store more than 1000 integers?

      resize – or error message

    - What if the user only needs 5 integers, and not 1000?

      wasteful

  - If you hardcode the value 1000 in the body of the program, this is not good. Why not?

    leads to bugs if not consistently changed

  - If you hardcode it as a symbolic constant, this is better; but, there's still a problem:

    change & recompile

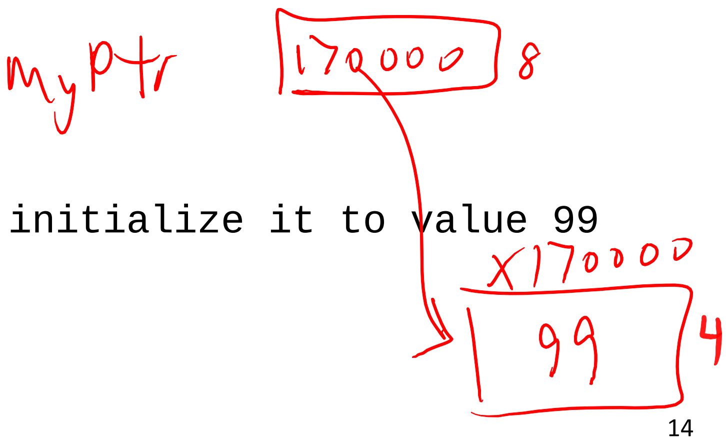
Dynamic Memory Allocation (cont.)

- Solution: // See also Koffman pages 26-27, 34-35, 197, etc.
  - When our program runs, we can request extra space on-the-fly (i.e., when we need it—even large amounts!) from the *free store* (memory heap).
  - We'll use two functions to handle our request for memory (called "allocation") from the heap, and our return of that memory (when we don't need it anymore—called "deallocation"):

1. Function `new` returns a *pointer* to a memory block for a given type:

```
// Allocate space for exactly one integer.
int *myPtr = new int;
*myPtr = 99;

// Alternatively:
int *myPtr = new int(99);    // initialize it to value 99
```

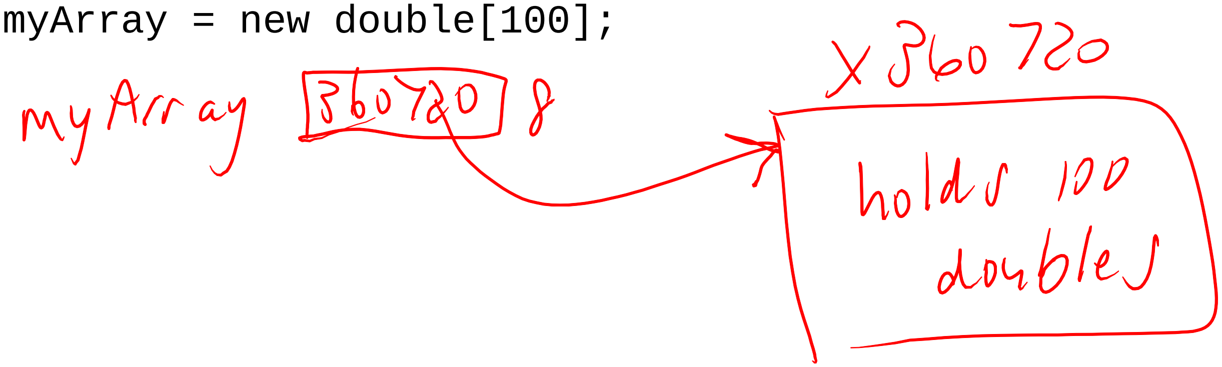


Dynamic Memory Allocation (cont.)

For arrays:

```
cout << “Enter the number of students: ”;
cin  >> num_students;
string * students = new string[num_students];

// Another example:  myArray points to the start of
// an array of 100 doubles.
double *myArray = new double[100];
```



Dynamic Memory Allocation (cont.)

2. Function `delete` returns the memory (previously allocated with `new`) and pointed to by `myPtr` to the memory heap:

```
delete myPtr;
```

- Yes, the computer remembers how many bytes need to be freed, provided you give it the correct address (`myPtr`).

- It’s important to free the memory, or you may eventually run out.

- For arrays, the syntax is a little different, and we must point to the start of the array (i.e., equivalent to its first element):

```
delete [] students;
delete [] myArray;
```

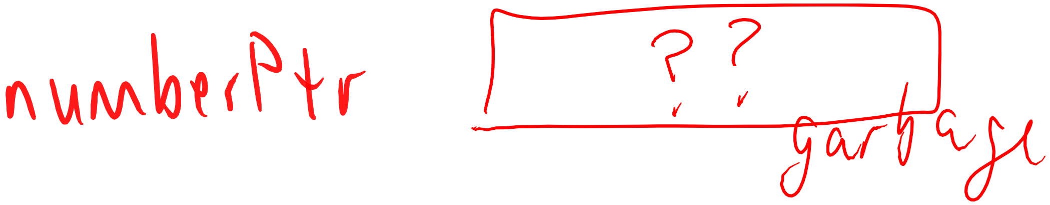
Addressing Errors (e.g., Segmentation Faults)

- Don’t try to dereference a pointer that isn’t pointing to a legitimate address for which you’ve previously called `new` or (in C) `malloc`.
  - This will usually crash the program, because you’re trying to dereference memory that doesn’t belong to you (e.g., the memory belongs to someone else, or it belongs to the operating system and is inaccessible to your program).

```
int * numberPtr;         // numberPtr is uninitialized

if (*numberPtr == NULL) // dereferencing garbage address,
                        // probably crashing program
```

- Also, if you’ve already freed memory, don’t try to re-access it.
- Also, don’t go out-of-bounds on an array or other data structure.



how_many	ptr
~~50003~~	~~4701260~~

Memory Leaks

- Keep track of the memory you allocate in a program; otherwise, you won't be able to reference it again! (or free it!)
  - Also, you might run out of memory, at some point. What's the problem with the following function?

50000

ptr ~~4701260~~ 8

```
void leakage(int how_many)
{
    int * ptr;
    
    ptr = new int[how_many];
  return;    // return statement is optional for void
}
```

The new operator allocates memory from the heap.

provides the starting address

- Exercise: Improve the above function, so that it's more useful.

x4701260

enough for 50000 ints

stays "around forever"

← even after leaving the function

Example of the Use of Pointers

Based on an example by Tharaja, Reema. *Data Structures Using C*. Oxford, pp. 109-110, and converted to C++. The value “25” gets printed by the cout statement.

```
void calculate_triangle_area(double * b, double * h, double * a);
// You can write double *b, etc., instead.

int main(void) {
  double  base, height, area;

  base = 10.0;
  height = 5.0;
  calculate_triangle_area(&base, &height,  &area);
  cout << "The area of the triangle is " << area << " square units." << endl;
  return 0;
}

void calculate_triangle_area(double * b, double * h, double * a) {
   *a = 0.5 * (*b) * (*h);     // parentheses are optional
}
```

x80010 base	x80008 height	x80000 area
10.0	5.0	? 25.0

0.5 10.0 5.0

b	h	a
80010	80008	80000

Example

double num; declaration

double& num

const double& num

In prototype or function declaration

Example of the Use of a Reference

Here is another example that also prints “25” as the output value:

```
void calculate_triangle_area(double b, double h, double& a);
// The ampersand (&) creates a reference.  The function can update the caller's memory.

int main(void) {
  double  base, height, area;

  base = 10.0;
  height = 5.0;
  calculate_triangle_area(base, height, area);
  cout << "The area of the triangle is " << area << " square units." << endl;

  return 0;
}

void calculate_triangle_area(double b, double h, double& a) {
   a = 0.5 * b * h;             // parentheses are optional
}
```

const means function can't change it, (but here we want to allow changes)

base	height	area
10.0	5.0	25.0

10.0 5.0

~~b~~	~~h~~	~~a~~
~~10.0~~	~~5.0~~	

Exercise: Change the Second Program to Avoid the Use of Pointers or References

What changes would you make, below, while still calling a function?

```
void calculate_triangle_area(double b, double h, ~~double& a~~);
// The ampersand (&) creates a reference.  The function can update the caller's memory.

int main(void) {
  double  base, height, area;

  base = 10.0;
  height = 5.0;
area=calculate_triangle_area(base, height, ~~area~~);
  cout << "The area of the triangle is " << area << " square units." << endl;

  return 0;
}
double
~~void~~ calculate_triangle_area(double b, double h, ~~double& a~~) {
   a = 0.5 * b * h;               // parentheses are optional
}
```