Eric Roberts
CS 106B

Handout #24
April 22, 2009

# Memory and C++

## Data Types in C++

Before we can talk about objects in C++, it is useful to review the more traditional data types that C++ inherits from C:

- Atomic types:
  - `short`, `int`, `long`, and their `unsigned` variants
  - `float`, `double`, and `long double`
  - `char`
  - `bool`
- Enumerated types defined using the `enum` keyword
- Structure types defined using the `struct` keyword
- Arrays of some base type
- Pointers to a target type

## Simple Arrays in C++

- We haven't actually used arrays in their low-level form this quarter, because the `Vector` class is so much better.
- From the client perspective, an array is like a brain-damaged form of `Vector` with the following differences:
  - The only operation is selection using `[]`
  - Array selection does not check that the index is in range
  - The length of an array is fixed at the time it is created
  - Arrays don't store their length, so programs that use them must pass an extra integer value that represents the **effective size**
- Array variables are declared using the following syntax:

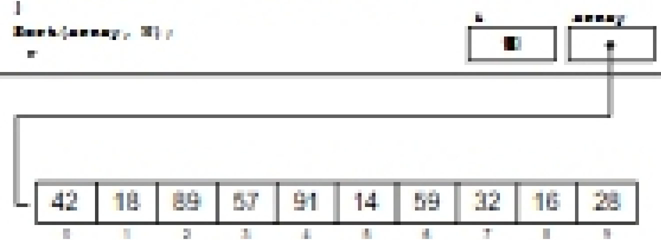
  *type name*[*n*] ;

  where *type* is the element type, *name* is the array name, and *n* is a constant integer expression indicating the length.

## A Simple Array Example

```
const int N = 10;

int main() {
   int array[N];
   for ( int i = 0 ; i < N ; i++ ) {
      array[i] = RandomInteger(0, 99);
   }
   Sort(array, N);
}
```




## The Structure of Memory

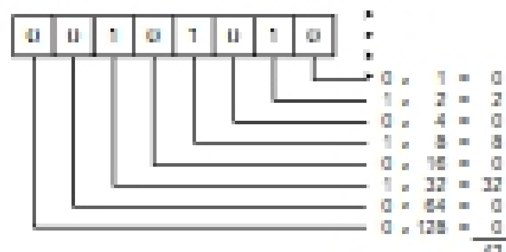
- The fundamental unit of memory inside a computer is called a **bit**, which is a contraction of the words *binary digit*. A bit can be in either of two states, usually denoted as 0 and 1.
- The hardware structure of a computer combines individual bits into larger units. In most modern architectures, the smallest unit on which the hardware operates is a sequence of eight consecutive bits called a **byte**. The following diagram shows a byte containing a combination of 0s and 1s:

| 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|

- Numbers are stored in still larger units that consist of multiple bytes. The unit that represents the most common integer size on a particular hardware is called a **word**. Because machines have different architectures, the number of bytes in a word may vary from machine to machine.

## Binary Notation

- Bytes and words can be used to represent integers of different sizes by interpreting the bits as a number in **binary notation**.
- Binary notation is similar to decimal notation but uses a different **base**. Decimal numbers use 10 as their base, which means that each digit counts for ten times as much as the digit to its right. Binary notation uses base 2, which means that each position counts for twice as much, as follows:




## Numbers and Bases

- The calculation at the end of the preceding slide makes it clear that the binary representation 00101010 is equivalent to the number 42. When it is important to distinguish the base, the text uses a small subscript, like this:

$$00101010_2 = 42_{10}$$

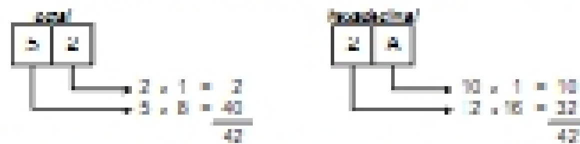
- Although it is useful to be able to convert a number from one base to another, it is important to remember that the number remains the same. What changes is how you write it down.
- The number 42 is what you get if you count how many stars are in the pattern at the right. The number is the same whether you write it in English as *forty-two*, in decimal as 42, or in binary as 00101010.

★★★★★★★
★★★★★★★
★★★★★★★
★★★★★★★
★★★★★★★
★★★★★★★

- Numbers do not have bases; representations do.

## Octal and Hexadecimal Notation

- Because binary notation tends to get rather long, computer scientists often prefer **octal** (base 8) or **hexadecimal** (base 16) notation instead. Octal notation uses eight digits: 0 to 7. Hexadecimal notation uses sixteen digits: 0 to 9, followed by the letters `A` through `F` to indicate the values 10 to 15.
- The following diagrams show how the number forty-two appears in both octal and hexadecimal notation:




- The advantage of using either octal or hexadecimal notation is that doing so makes it easy to translate the number back to individual bits because you can convert each digit separately.

## Exercises: Number Bases

- What is the decimal value for each of the following numbers?

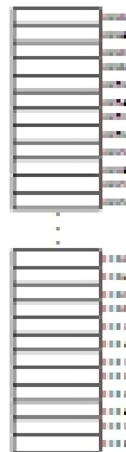
  $10001_2$ $177_8$ $AD_{16}$

- As part of a code to identify the file type, every Java class file begins with the following sixteen bits:

| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

  How would you express that number in hexadecimal notation?

## Memory and Addresses



- Every byte inside the primary memory of a machine is identified by a numeric address. The addresses begin at 0 and extend up to the number of bytes in the machine, as shown in the diagram on the right.
- Memory diagrams that show individual bytes are not as useful as those that are organized into words. The revised diagram on the right now includes four bytes in each of the memory cells, which means that the address numbers increase by four each time.
- In these slides, addresses are four-digit hexadecimal numbers, which makes them easy to recognize.
- When you create memory diagrams, you don't know the actual memory addresses at which values are stored, but you do know that everything has an address. **Just make something up.**

## Sizes of the Fundamental Types

- The memory space required to represent a value depends on the type of value. Although the C++ standard actually allows compilers some flexibility, the following sizes are typical:

| 1 byte (8 bits) | 2 bytes (16 bits) | 4 bytes (32 bits) | 8 bytes (64 bits) | 16 bytes (128 bits) |
|---|---|---|---|---|
| `char` `bool` | `short` | `int` `float` | `long` `double` | `long double` |

- Enumerated types are typically assigned the space of an `int`.
- Structure types have a size equal to the sum of their fields.
- Arrays take up the element size times the number of elements.
- Pointers take up the space needed to hold an address, which is 4 bytes on a 32-bit machine and 8 bytes on a 64-bit machine.
- The expression `sizeof(t)` returns the size of the type *t*.

## Pointers

- In C++, every value is stored somewhere in memory and can therefore be identified with that address. Such addresses are called ***pointers***.
- Because C++ is designed to allow programmers to control data at the lowest level, pointers can be manipulated just like any other kind of data. In particularly, you can assign one pointer value to another, which means that the two pointers end up indicating the same data value.
- Diagrams that involve pointers are typically represented in two different ways. Using memory addresses emphasizes the fact that pointers are just like integers. Conceptually, it often makes more sense to respresent a pointer as an arrow. The head of the arrow is positioned at the address in memory at which the object lives. The tail of the arrow is positioned inside the variable that holds the pointer value.

## Declaring a Pointer Variable

- Pointer variables have a declaration syntax that may at first seem confusing. To declare a variable as a pointer to a particular type as opposed to a variable of that type, all you need to do is add a `*` in front of the variable name, like this:

```
type *var;
```

- For example, if you wanted to declare a variable `px` to be a pointer to a `double` value, you could do so as follows:

```
double *px;
```

  Similarly, to declare a variable `pptr` as a pointer to a `pointT` structure, you would write:

```
pointT *pptr;
```

## Pointer Operators

- C++ includes two built-in operators for working with pointers:
  - The address-of operator (`&`) is written before a variable name (or any expression to which you could assign a value) and returns the address of that variable. Thus, the expression `&total` gives the the address of `total` in memory.
  - The deference operator (`*`) is written before a pointer expression and returns the actual value to which the pointer points.
- Suppose, for example, that you have declared and initialized the following variables:

```
double x = 2.5;
double *px = &x;
```

- At this point, the variable `px` points to the variable `x`, and the expression `*px` is synonymous with the variable `x`.

## Pointers and Arrays

- In C++, all arrays are represented internally as a pointer to their first element.
- For example, if you declare an array

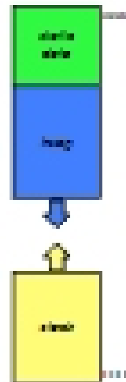
```
int list[100];
```

  the C++ compiler treats the name `list` as a shorthand for the expression `&list[0]`.

- You can freely intermix array and pointer notation in your code. If you declare something as an array of a particular type, you can use it as a pointer to that type, and vice versa.

## The Allocation of Memory to Variables

- When you declare a variable in a program, C++ allocates space for that variable from one of several memory regions.
- One region of memory is reserved for variables that persist throughout the lifetime of the program, such as constants. This information is called **static data**.
- Each time you call a method, C++ allocates a new block of memory called a **stack frame** to hold its local variables. These stack frames come from a region of memory called the **stack**.
- It is also possible to allocate memory dynamically, as described on the next slide. This space comes from a pool of memory called the **heap**.
- In classical architectures, the stack and heap grow toward each other to maximize the available space.



## Dynamic Allocation

- Because it is a hybrid of C and a variety of object-oriented features, C++ offers two distinct mechanisms for allocating memory on the heap.
- The first style uses the function `malloc(size)` to create a block of `size` bytes. This style is used primarily to create dynamic array space. If, for example, you needed to allocate space for an array of ten million integers, you would write

```
int *array = malloc(10000000 * sizeof(int));
```

- The second form uses the `new` operator with a type to allocate space for a value of that type. For example, to allocate space for a `pointT` structure on the heap, you would write

```
pointT *ptr = new pointT;
```

- Space allocated using `malloc` is freed by calling `free`; space allocated using `new` is freed by invoking the `delete` operator.

## Heap-Stack Diagrams

- It is easier to understand how C++ works if you have a good mental model of its use of memory. I find the most useful model is a **heap-stack diagram**, which shows the heap on the left and the stack on the right, separated by a dotted line.
- Whenever your program uses `malloc` or `new`, you need to add a block of memory to the heap side of the diagram. That block must be large enough to store the entire value you're allocating. If the value is a `struct` or an object type (which we'll talk about beginning on Friday), that block must include space for all the members inside that structure.
- Whenever your program calls a method, you need to create a new stack frame by adding a block of memory to the stack side. For method calls, you need to add enough space to store the local variables for the method, again with some overhead information that tracks what the program is doing. When a method returns, C++ reclaims the memory in its frame.

## Exercise: Heap-Stack Diagrams

Trace the evolution of the heap and stack in the execution of the following program:

```
int main() {
   pointT pt;
   double total = 0.0;
   pt.x = 1;
   pt.y = 2;
   int *array = malloc(3 * sizeof(int));
   Nonsense(array, pt, total);
   return 0;
}

void Nonsense(int list[], pointT pt, double & total) {
   pointT *pptr = new pointT;
   list[1] = pt.x;
   total += pt.y;
}
```