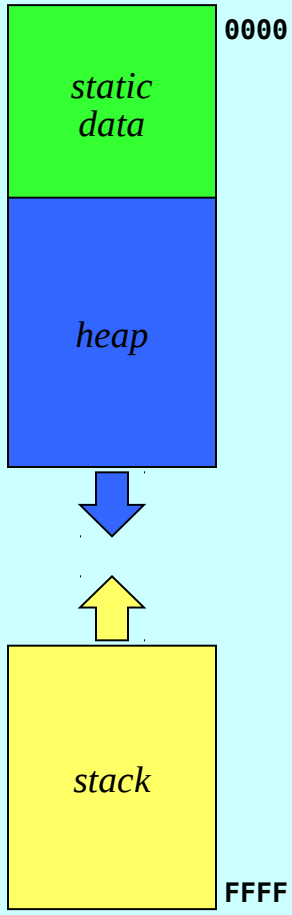
Dynamic Allocation

Eric Roberts
CS 106B
February 2, 2015

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 in Chapter 12. 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

- C++ uses the **`new`** operator to allocate memory on the heap.
- You can allocate a single value (as opposed to an array) by writing **`new`** followed by the type name. Thus, to allocate space for a **`int`** on the heap, you would write

```
int *ip = new int;
```

- You can allocate an array of values using the following form:

`new` *type*`[`*size*`]`

Thus, to allocate an array of 10000 integers, you would write:

```
int *array = new int[10000];
```

- The **`delete`** operator frees memory previously allocated. For arrays, you need to include empty brackets, as in

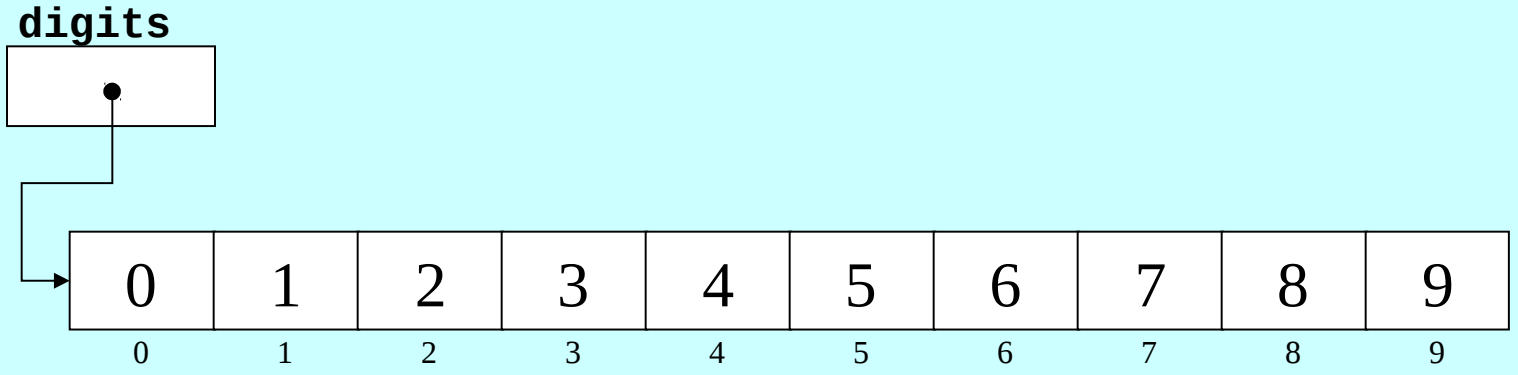
```
delete[] array;
```

Exercise: Dynamic Arrays

- Write a method **`createIndexArray(n)`** that returns an integer array of size **`n`** in which each element is initialized to its index. As an example, calling

```
int *digits = createIndexArray(10);
```

should result in the following configuration:

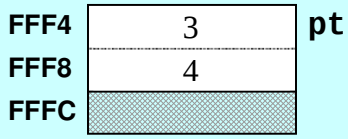


- How would you free the memory allocated by this call?

Allocating a **`Point`** Object

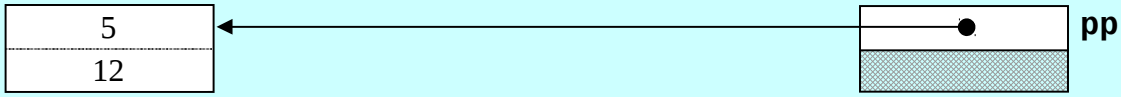
- The usual way to allocate a **`Point`** object is to declare it as a local variable on the stack, as follows:

```
Point pt(3, 4);
```



- It is, however, also possible to allocate a **`Point`** object on the heap using the following code:

```
Point *pp = new Point(5, 12);
```



The `->` Operator

- In C++, pointers are explicit. Given a pointer to an object, you need to dereference the pointer before selecting a field or calling a method. Given the definition of **`pp`** from the previous slide, you cannot write

```
pp.getX()
```

  because **`pp`** is not a structure.

- You also cannot write

```
*pp.getX()
```

  because `.` takes precedence over `*`.

- To call a method given a pointer to an object, you need to write

```
pp->getX()
```

The Keyword **`this`**

- In the implementation of the methods within a class, you can usually refer to the private instance variables of that class using just their names. C++ resolves such names by looking for matches in each of the following categories:
  - Parameters or local variables declared in the current method
  - Instance variables of the current object
  - ~~Global variables defined in this scope~~
- It is often convenient to use the same names for parameters and instance variables. If you do, you must use the keyword **`this`** (defined as a pointer to the current object) to refer to the instance variable, as in the constructor for the **`Point`** class:

```
Point::Point(int x, int y) {
   this->x = x;
   this->y = y;
}
```

Memory Management

- The big challenge in working with dynamic allocation is freeing the heap memory you allocate. Programs that fail to do so have what computer scientists call ***memory leaks***.
- In Java, objects are created on the heap and are automatically reclaimed by a ***garbage collector*** when those objects are no longer accessible. This discipline makes memory management very easy and convenient.
- In C++, the situation is different for the following reasons:
  - Objects can be allocated either on the stack or in the heap.
  - C++ has no garbage collector, which means that programmers must manage memory allocation explicitly.
- Fortunately, the designers of C++ made it possible to free memory allocated by objects when those objects go out of scope on the stack. . . .

Destructors

- In C++, class definitions often include a ***destructor***, which specifies how to free the storage used to represent an instance of that class.
- The prototype for a destructor has no return type and uses the name of the class preceded by a tilde (**~**). The destructor must not take any arguments.
- C++ calls the destructor automatically whenever a variable of a particular class is released. For stack objects, this happens when the function returns. The effect of this rule is that a C++ program that declares its objects as local variables on the stack will automatically reclaim those variables.
- If you instead allocate space for an object in the heap using **`new`**, you must explicitly free that object by calling **`delete`**. Calling **`delete`** automatically invokes the destructor.

The `CharStack` Class

`CharStack cstk;` Initializes an empty stack.
`cstk.size()` Returns the number of characters pushed onto the stack.
`cstk.isEmpty()` Returns **`true`** if the stack is empty.
`cstk.clear()` Deletes all characters from the stack.
`cstk.push(ch)` Pushes a new character onto the stack.
`cstk.pop()` Removes and returns the top character from the stack.

The `charstack.h` Interface

```
/*
 * File: charstack.h
 * -----------------
 * This interface defines the CharStack class.
 */

#ifndef _charstack_h
#define _charstack_h

class CharStack {
public:

/*
 * CharStack constructor and destructor
 * ------------------------------------
 * The constructor initializes an empty stack.  The destructor
 * is responsible for freeing heap storage.
 */

   CharStack();
   ~CharStack();
```

The `charstack.h` Interface

```
/*
 * Methods: size, isEmpty, clear, push, pop
 * ----------------------------------------
 * These methods work exactly as they do for the Stack class.
 * The peek method has been eliminated to save space.
 */

   int size();
   bool isEmpty();
   void clear();
   void push(char ch);
   char pop();
```

The `charstack.h` Interface

```
/* Private section */

private:

/* Private constants */

   static const int INITIAL_CAPACITY = 10;

/* Instance variables */

   char *array;           /* Dynamic array of characters   */
   int capacity;          /* Allocated size of that array  */
   int count;             /* Current count of chars pushed */

/* Private function prototype */

   void expandCapacity();

};

#endif
```

The `charstack.cpp` Implementation

```
/*
 * File: charstack.cpp
 * -------------------
 * This file implements the CharStack class.
 */

#include "charstack.h"
#include "error.h"
using namespace std;

/* Constructor and destructor */

CharStack::CharStack() {
   capacity = INITIAL_CAPACITY;
   array = new char[capacity];
   count = 0;
}

CharStack::~CharStack() {
   delete[] array;
}
```

The **charstack.cpp** Implementation

```
int CharStack::size() {
   return count;
}

bool CharStack::isEmpty() {
   return count == 0;
}

void CharStack::clear() {
   count = 0;
}

void CharStack::push(char ch) {
   if (count == capacity) expandCapacity();
   array[count++] = ch;
}

char CharStack::pop() {
   if (isEmpty()) error("pop: Stack is empty");
   return array[--count];
}
```

The **charstack.cpp** Implementation

```
/*
 * Implementation notes: expandCapacity
 * ------------------------------------
 * This private method doubles the capacity of the elements array
 * whenever it runs out of space.  To do so, it must copy the
 * pointer to the old array, allocate a new array with twice the
 * capacity, copy the characters from the old array to the new
 * one, and finally free the old storage.
 */

void CharStack::expandCapacity() {
   char *oldArray = array;
   capacity *= 2;
   array = new char[capacity];
   for (int i = 0; i < count; i++) {
      array[i] = oldArray[i];
   }
   delete[] oldArray;
}
```

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 **`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, 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

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

heap

???	1000
1	1004
???	1008
???	100C
???	1010
???	1014
???	1018

stack

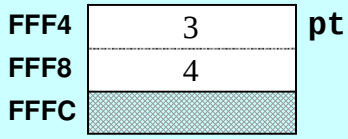
list	1000	FFD0
pt	1	FFD4
	2	FFD8
total &	FFF4	FFDC
pptr	1014	FFE0
		FFE4
array	1000	FFE8
pt	1	FFEC
	2	FFF0
total	2.0	FFF4
		FFF8
		FFFC

The End

Allocating a **`Point`** Object

- The usual way to allocate a **`Point`** object is to declare it as a local variable on the stack, as follows:

```
Point pt(3, 4);
```



- It is, however, also possible to allocate a **`Point`** object on the heap using the following code:

```
Point *pp = new Point(5, 12);
```