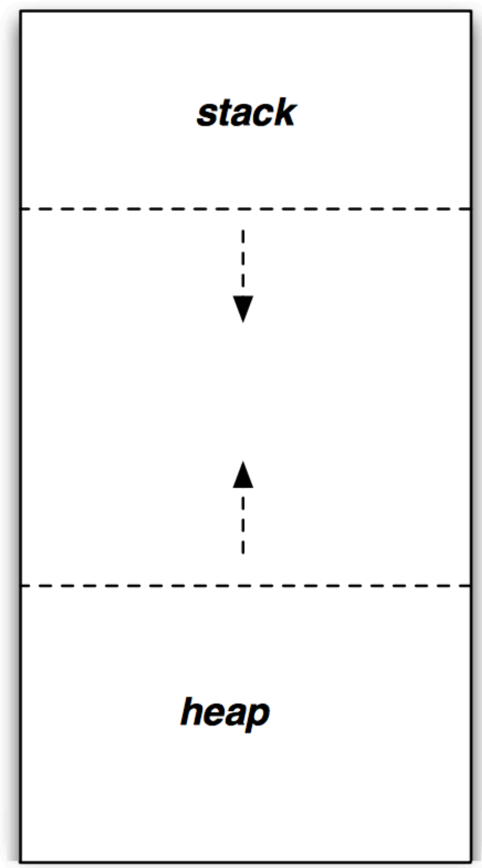
CS 106X

Lecture 15: Dynamic Memory Allocation

Monday, February 13, 2017



Programming Abstractions (accelerated)
Fall 2016
Stanford University
Computer Science Department

Lecturer: Chris Gregg

reading:
Programming Abstractions in C++, Chapter 11

Today's Topics

- Logistics
  - Midterm information:
    - Thursday February 23rd
    - Midterm Review Session: TBA
    - There will be practice midterms on the website.
    - Tiny Feedback: "Is there any other interesting youtube channel you can recommend besides Numberphile?" **Glad you asked! (see YouTube Video page at the end of the slides)**

- More on Pointers
  - Operator overloading recap
  - Mystery function
  - Pointers and Structs
  - The -> operator
- Dynamic Memory Allocation
  - The new and delete keywords
  - The "heap"

Recap of Pointer Syntax from the last lecture

- Pointer Syntax #1:
  - To declare a pointer, use the ***** symbol. Example:

```
string *laserPtr = NULL; // creates a pointer to a string and
                         // sets it to point to a non-usable address.
```

- Pointer Syntax #2:
  - To put a variable's address into a pointer, use the **&** symbol. Example:

```
string laser = "green"; // has some address (example: 0x12a5)
laserPtr = &laser; // puts the address of laser
                   // into the laserPtr variable.
                   // The value of laserPtr is now 0x12a5
```

- Pointer Syntax #3:

  To *get value of the variable a pointer points to*, use the "*****". Example:

```
string laserCopy = *laserPtr; // laserCopy is now "green"
```

Operator Overloading: the What and the Why

- More Tiny Feedback: "*Not sure I understood operator overriding [sic] very well.*"

- C++ allows us to *overload* operators. What does that mean?
  - It means that if we create a class, we can utilize common operators (e.g., +, *, <<, etc.) to do what we want.

- Why do we do it?
  - We want to make things easier and more straightforward for the user of our class.

Operator Overloading: Example

- Let's look back at our Fraction class. We implemented a mult() function:

```
void Fraction::mult(Fraction other)
{
        // multiplies a Fraction
        // with this Fraction
        num *= other.num;
        denom *= other.denom;

        // reduce the fraction
        reduce();
}
```

- But, this isn't the normal way we multiply numbers. We can fix that!

Operator Overloading: Example

- Our .h file needs a friend function:

```
class Fraction {
public:
                                    ...

    friend Fraction operator*(const Fraction &first, const Fraction &second);

                                    ...
```

- The "const" just means we aren't allowed to change either number when multiplying.

Operator Overloading: Example

- The function in our .cpp file:

```
Fraction operator*(const Fraction &first, const Fraction &second) {
    int newNum = first.num * second.num;
    int newDenom = first.denom * second.denom;

    return Fraction(newNum,newDenom); // will be reduced automatically
}
```

- We simply return a new Fraction (which copies its elements. Luckily, they aren't pointers!)
- We call the function like this:

```
Fraction two_thirds = Fraction(2,3);
Fraction three_fourths = Fraction(3,4);
Fraction result = two_thirds * three_fourths; // result = 1/2
```

overloaded *

Introduction to Pointers

What is a pointer??

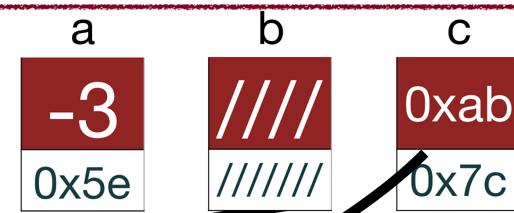
a memory address!

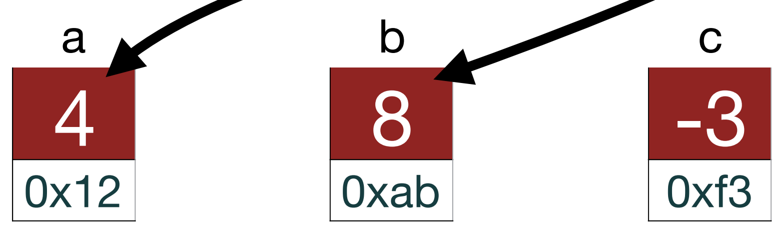
Mystery Function: What prints out?

```
void mystery(int a, int& b, int* c) {
    a++;
    (*c)--;
    b += *c;
    cout << a << " " << b << " " << *c << " " << endl;
}

int main() {
    int a = 4;
    int b = 8;
    int c = -3;
    cout << a << " " << b << " " << c << " " << endl;
    mystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
```

Mystery Function: What prints out?

```
void mystery(int a, int& b, int* c) {
    a++;
    (*c)--;
    b += *c;
    cout << a << " " << b << " " << *c << " " << endl;
}

int main() {
    int a = 4;
    int b = 8;
    int c = -3;
    cout << a << " " << b << " " << c << " " << endl;
    mystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
```



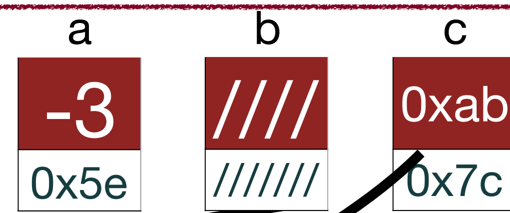


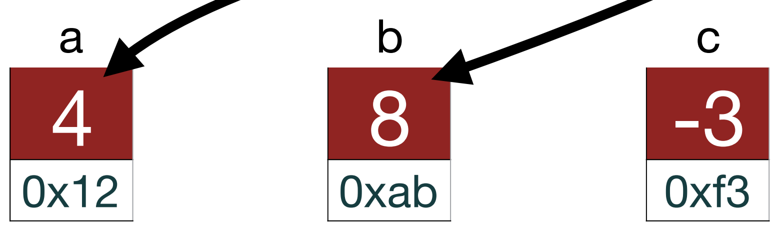
Answer:

4 8 -3

Mystery Function: What prints out?

```
void mystery(int a, int& b, int* c) {
    a++;
    (*c)--;
    b += *c;
    cout << a << " " << b << " " << *c << " " << endl;
}

int main() {
    int a = 4;
    int b = 8;
    int c = -3;
    cout << a << " " << b << " " << c << " " << endl;
    mystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
```





Answer:

4 8 -3

Mystery Function: What prints out?

```
void mystery(int a, int& b, int* c) {
    a++;
    (*c)--;
    b += *c;
    cout << a << " " << b << " " << *c << " " << endl;
}

int main() {
    int a = 4;
    int b = 8;
    int c = -3;
    cout << a << " " << b << " " << c << " " << endl;
    mystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
```

a	b	c
-2	////	0xab
0x5e	///////	0x7c

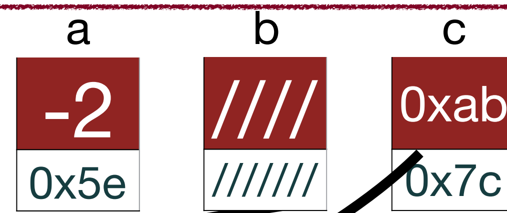
a	b	c
4	8	-3
0x12	0xab	0xf3

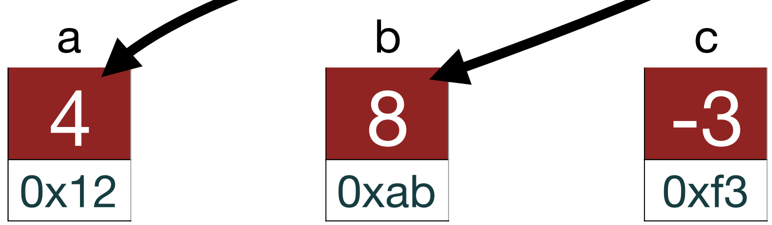
Answer:

4 8 -3

Mystery Function: What prints out?

```
void mystery(int a, int& b, int* c) {
    a++;
    (*c)--;
    b += *c;
    cout << a << " " << b << " " << *c << " " << endl;
}

int main() {
    int a = 4;
    int b = 8;
    int c = -3;
    cout << a << " " << b << " " << c << " " << endl;
    mystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
```



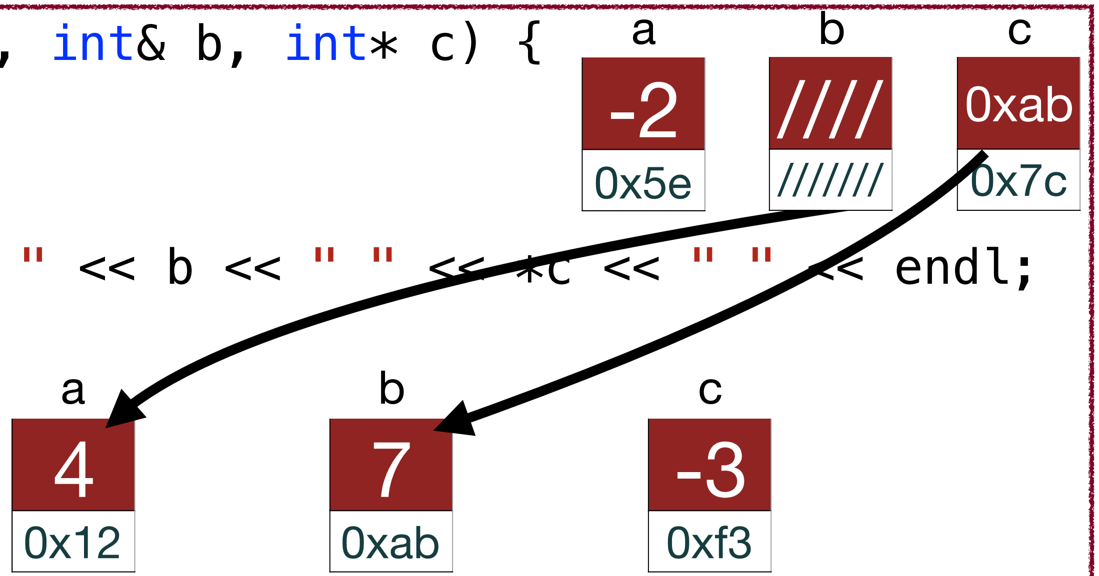


Answer:

4 8 -3

Mystery Function: What prints out?

```
void mystery(int a, int& b, int* c) {
    a++;
    (*c)--;
    b += *c;
    cout << a << " " << b << " " << *c << " " << endl;
}

int main() {
    int a = 4;
    int b = 8;
    int c = -3;
    cout << a << " " << b << " " << c << " " << endl;
    mystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
```



Answer:

4 8 -3

Mystery Function: What prints out?

```
void mystery(int a, int& b, int* c) {
    a++;
    (*c)--;
    b += *c;
    cout << a << " " << b << " " << *c << " " << endl;
}

int main() {
    int a = 4;
    int b = 8;
    int c = -3;
    cout << a << " " << b << " " << c << " " << endl;
    mystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
```

a	b	c
-2	////	0xab
0x5e	///////	0x7c

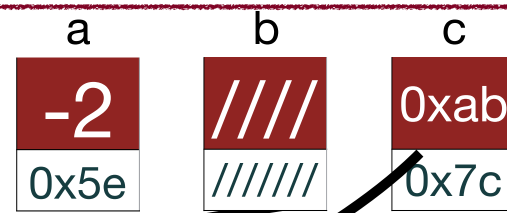
a	b	c
4	7	-3
0x12	0xab	0xf3

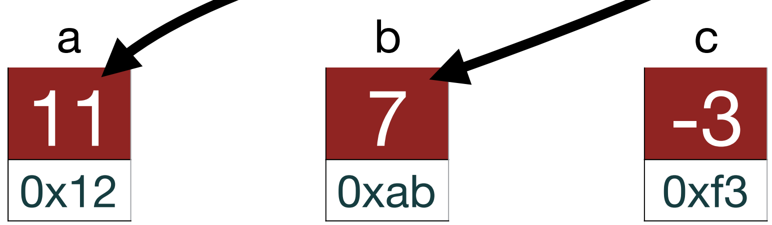
Answer:

4 8 -3

Mystery Function: What prints out?

```
void mystery(int a, int& b, int* c) {
    a++;
    (*c)--;
    b += *c;
    cout << a << " " << b << " " << *c << " " << endl;
}

int main() {
    int a = 4;
    int b = 8;
    int c = -3;
    cout << a << " " << b << " " << c << " " << endl;
    mystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
```



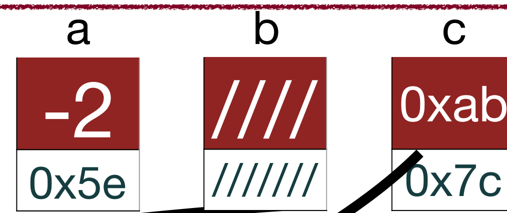


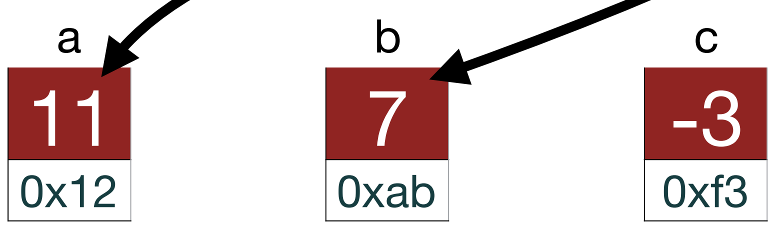
Answer:

4 8 −3

Mystery Function: What prints out?

```
void mystery(int a, int& b, int* c) {
    a++;
    (*c)--;
    b += *c;
    cout << a << " " << b << " " << *c << " " << endl;
}

int main() {
    int a = 4;
    int b = 8;
    int c = -3;
    cout << a << " " << b << " " << c << " " << endl;
    mystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
```



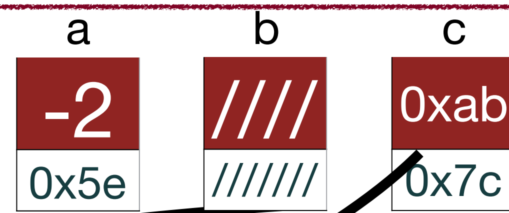


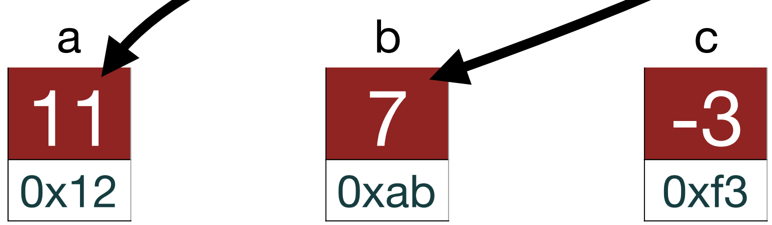
Answer:

4 8 -3

Mystery Function: What prints out?

```
void mystery(int a, int& b, int* c) {
    a++;
    (*c)--;
    b += *c;
    cout << a << " " << b << " " << *c << " " << endl;
}

int main() {
    int a = 4;
    int b = 8;
    int c = -3;
    cout << a << " " << b << " " << c << " " << endl;
    mystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
```



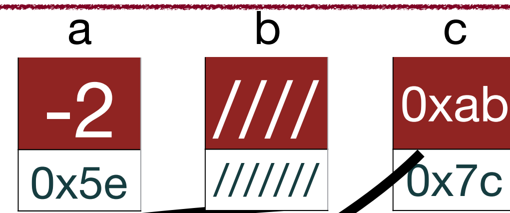


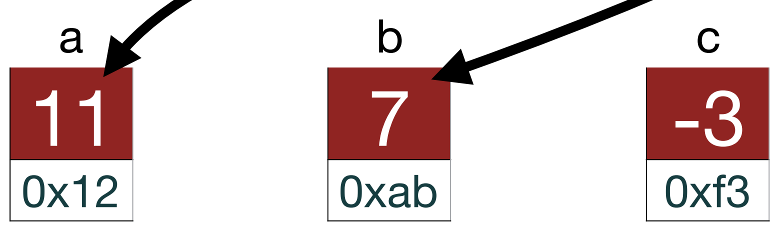
Answer:

```
 4   8  -3
-2  11   7
```

Mystery Function: What prints out?

```
void mystery(int a, int& b, int* c) {
    a++;
    (*c)--;
    b += *c;
    cout << a << " " << b << " " << *c << " " << endl;
}

int main() {
    int a = 4;
    int b = 8;
    int c = -3;
    cout << a << " " << b << " " << c << " " << endl;
    mystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
```



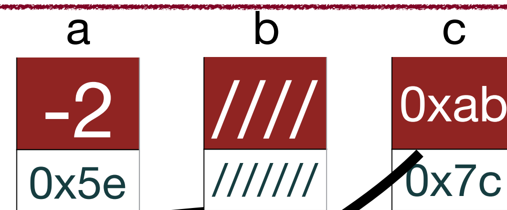


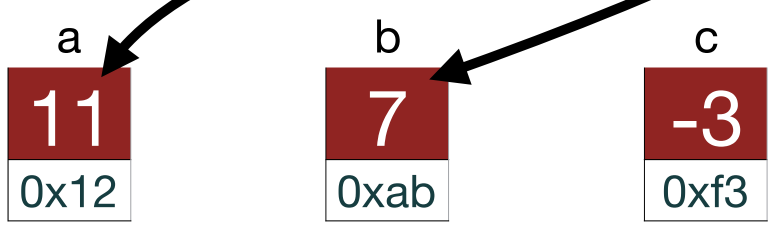
Answer:

```
 4    8   -3
-2   11    7
```

Mystery Function: What prints out?

```
void mystery(int a, int& b, int* c) {
    a++;
    (*c)--;
    b += *c;
    cout << a << " " << b << " " << *c << " " << endl;
}
```






```
int main() {
    int a = 4;
    int b = 8;
    int c = -3;
    cout << a << " " << b << " " << c << " " << endl;
    mystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
```

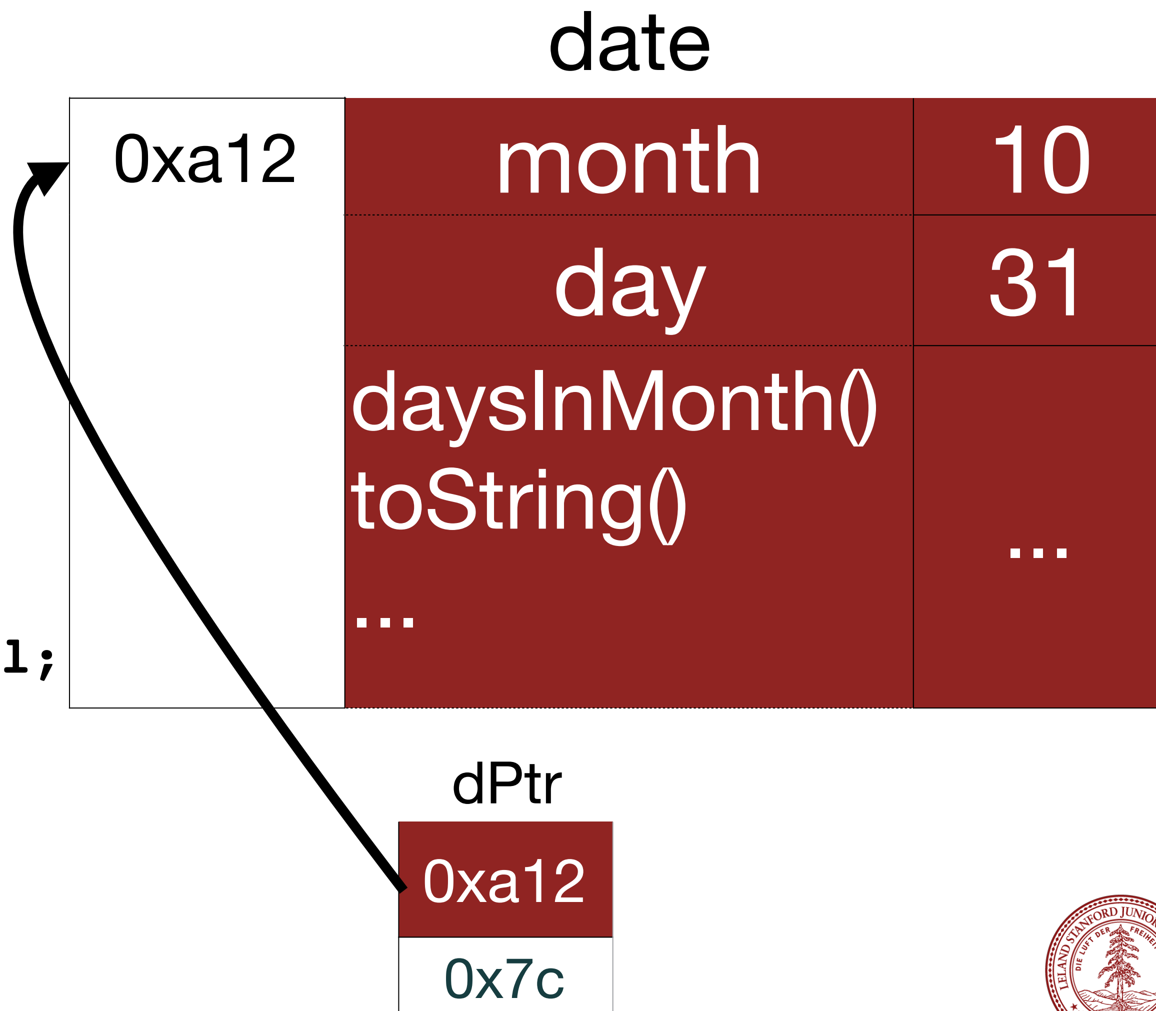
Answer:

4	8	-3
-2	11	7
11	7	-3

Pointers and Structs

- Pointers can point to a struct or class instance as well as to a regular variable.
- One way to do this would be to dereference and then use dot notation:

```
Date d;
d.month = 7;
Date* dPtr = &d;
cout << (*dPtr).month << endl;
```

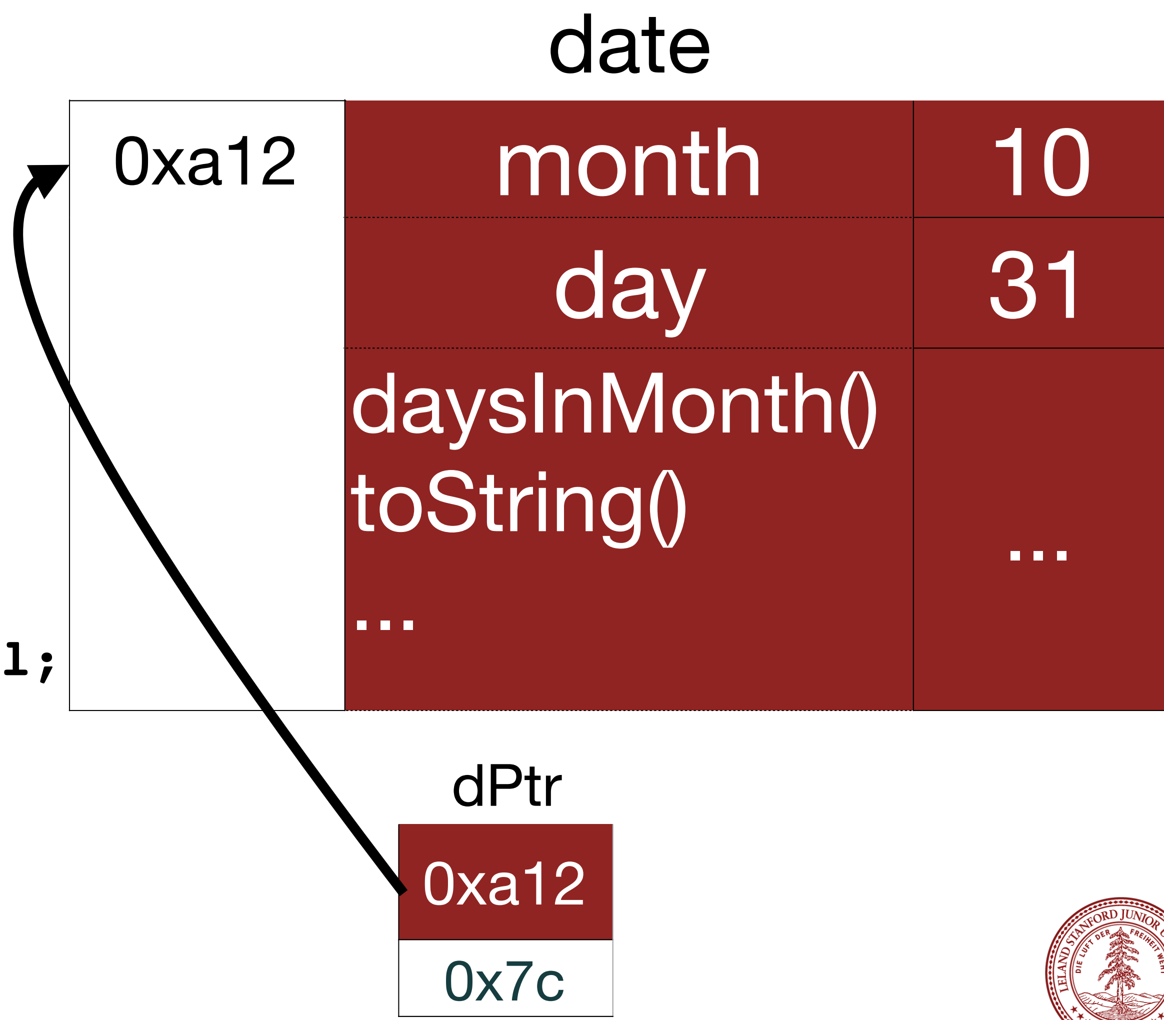


Pointers and Structs

- Pointers can point to a struct or class instance as well as to a regular variable.
- One way to do this would be to dereference and then use dot notation:

```
Date d;
d.month = 7;
Date* dPtr = &d;
cout << (*dPtr).month << endl;
```

- But, this notation is cumbersome, and the parenthesis are necessary because the "dot" has a higher precedence than the *****.

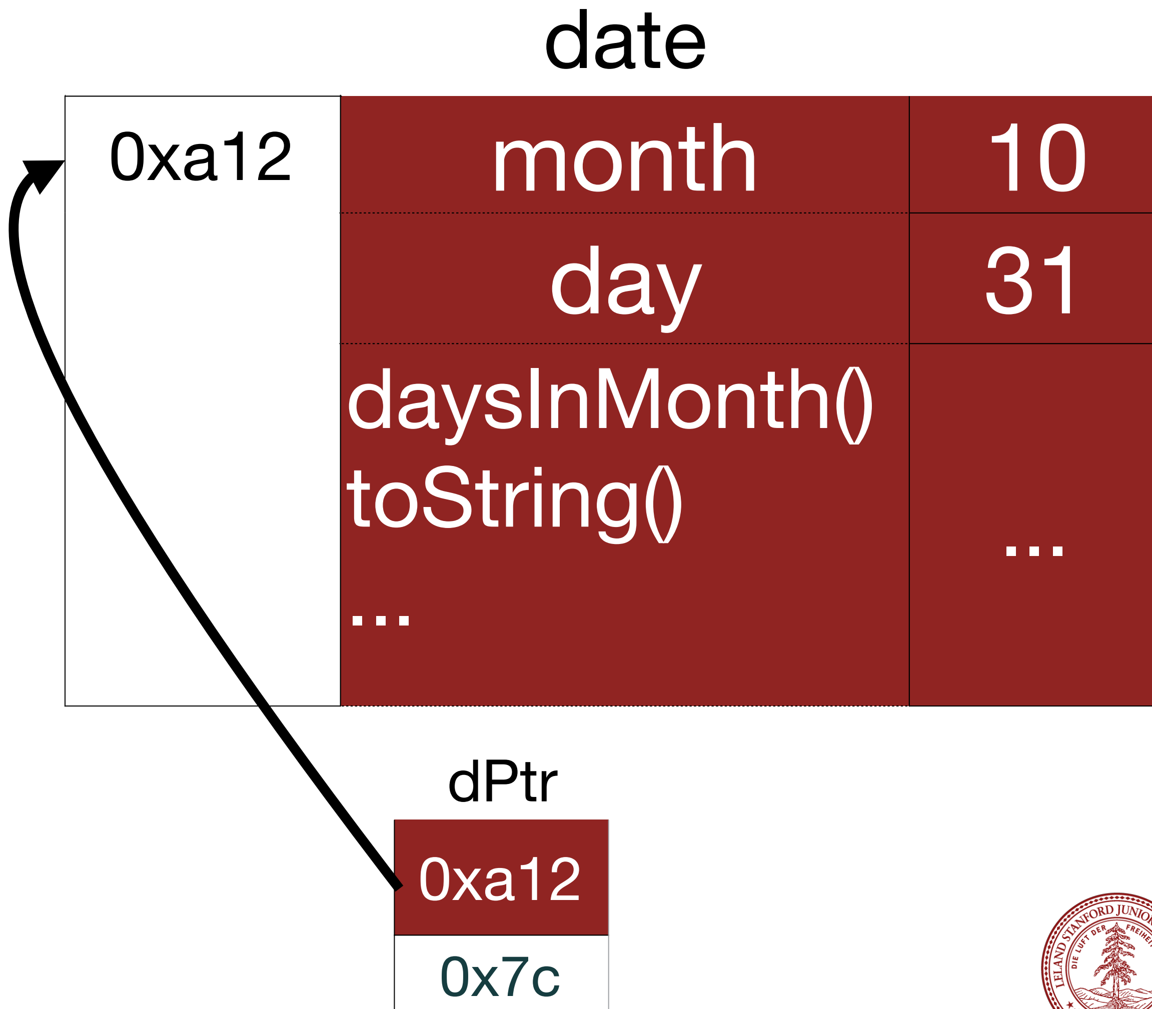


Pointers and Structs

- So, we have a different, and more intuitive syntax, called the "arrow" syntax, **->** :

```
Date d;
d.month = 7;
Date* dPtr = &d;
cout << dPtr->month << endl;
```

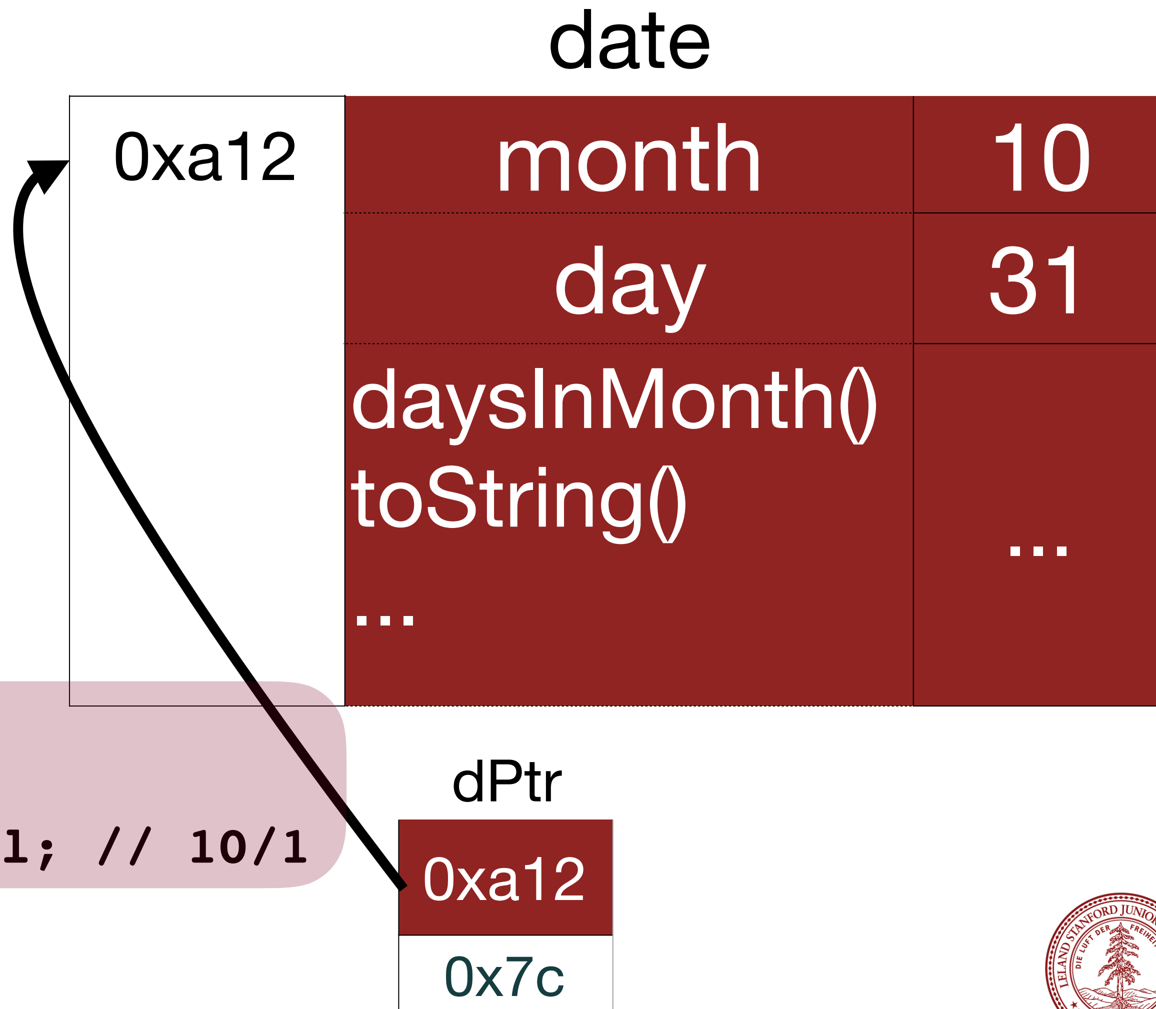
- We will use the arrow syntax almost exclusively when using structs.



Pointers and Structs

- The arrow syntax can be used to set a value or call a function in a struct via a pointer, as well:

```
Date d;
d.month = 7;
Date* dPtr = &d;
cout << dPtr->month << endl;
p->day = 1;
cout << p->day << endl; // 1
cout << p->toString() << endl; // 10/1
```



Dynamic Memory Allocation

- So far in this class, all variables we have seen have been local variables that we have defined inside functions. Sometimes, we have had to pass in object references to functions that modify those objects. For instance, take a look at the following code:

```
void squares(Vector<int> &vec, int numSquares) {
    for (int i=0; i < numSquares; i++) {
         vec.add(i * i);
    }
}
```

- This function requires the calling function to create a vector to use inside the function. This isn't necessarily bad, but could we do it a different way? In other words, could we create the Vector inside the function and just pass it back?

Dynamic Memory Allocation

- Could we create the Vector inside the function and just pass it back?

```
Vector<int> squares(int numSquares) {
    Vector<int> vec;
    for (int i=0; i < numSquares; i++) {
        vec.add(i * i);
    }
    return vec;
}
```

- Does this work?
  - It actually does, but there is an issue — you have to make a copy of the Vector, which is inefficient (though...these days if the creator of the Vector class is clever, we won't have to make a copy). Remember, we would rather not pass around large objects.

Dynamic Memory Allocation

- Okay…maybe we can do this?

```
Vector<int> &squares(int numSquares) {
    Vector<int> vec;
    for (int i=0; i < numSquares; i++) {
         vec.add(i * i);
    }
    return vec;
}
```

- Does this work?
  - No :(This is actually really bad. Why? The scope of **`vec`** is *only* the function, and you are not allowed to pass back a reference to a variable that goes out of scope.

Dynamic Memory Allocation

- Well, how about with pointers? Can we do this?

```
Vector<int> *squares(int numSquares) {
    Vector<int> vec;
    for (int i=0; i < numSquares; i++) {
        vec.add(i * i);
    }
    return &vec;
}
```

- Does this work?
    - No :(This is *also* really bad. Why? Same as before: the scope of **`vec`** is *only* the function, and you are not allowed to pass back a pointer to a variable that goes out of scope. When the function ends, the variable is destroyed, and your program will almost certainly crash.

Dynamic Memory Allocation

- What do we want here? What's the big deal?

- What we really want is really two things:

1. a way to reserve a section of memory so that it remains available to us *throughout our entire program*, or until we want to destroy it (give it back to the operating system)
2. a way to reserve any amount of memory we want at the time we need it.

- You might think that global variables are what we want, but that would be incorrect.
- Global variables can be accessed by any function in our program, and that isn't what we want. Also, global variables have a fixed size at compile time, and that isn't what we want, either.

Dynamic Memory Allocation: new

- C++ allows you to request memory from the operating system using the keyword **`new`**. This memory comes from the "heap" whereas variables you simply declare come from the "stack." Both of those terms will become important in CS 107, but for now, you need to know this:

  - Variables on the stack have a scope based on the function they are declared in.
  - Memory from the heap is allocated to your program **from the time you request the memory until the time you tell the operating system you no longer need it, or until your program ends.**

- To request memory from the heap, we use the following syntax:

```
type *variable = new type; // allocate one element
```

or

```
type *variable = new type[n]; // allocate n elements
```

Dynamic Memory Allocation: new

- Examples:

```
int *anInteger = new int; // create one integer on the heap

int *tenInts = new int[10]; // create 10 integers on the heap
```

- The second example (**`tenInts`**) is very powerful — the memory you are given is an array guaranteed by the operating system to be contiguous. So, that's how we allocate an array of items dynamically!

- Notice that **`new`** returns a pointer to the type you request — this is important! This is why we need to learn about pointers — in order to dynamically allocate memory, you have to use a pointer.

Arrays

- We have been using Vectors in class so far, and we've said "Oh, a Vector is just built on top of an array." So, let's talk about arrays for a bit. They are "lower level" than Vectors, and they are more limited.

```
int firstArray[10]; // create a static array on the stack;
                    // size of 10 is known at compile time

int *secondArray = new int[10]; // create 10 integers on the
                                // heap. Dynamically allocated.
// fill the arrays with values
for (int i=0; i < 10; i++) {
    firstArray[i] = i*2; // evens
    secondArray[i] = i*2 + 1; // odds
}
```

- Arrays are *not* objects, and they don't have functions, so there isn't any function like **`firstArray.length()`**. You have to keep track of the length!

Arrays

- **You have to keep track of the length!**

```
const int arrayLen = 10;

int firstArray[arrayLen]; // create a static array on the stack;
                          // size of 10 is known at compile time

int *secondArray = new int[arrayLen]; // create 10 integers on the
                                      // heap. Dynamically allocated.
// fill the arrays with values
for (int i=0; i < arrayLen; i++) {
    firstArray[i] = i*2; // evens
    secondArray[i] = i*2 + 1; // odds
}
```

- Notice, by the way, that we access our arrays by using bracket notation:
  - **`firstArray[i]`** gives us the value at index **`i`** in the array.

Arrays

- Unlike a vector, you can't just add another element past the end -- you are limited to the amount you asked for.

```
const int arrayLen = 10;

int *myArray = new int[arrayLen]; // create 10 integers on the
                                  // heap. Dynamically allocated.
// fill the array with values
for (int i=0; i < arrayLen; i++) {
   secondArray[i] = i*2 + 1; // odds
}

// add another?
secondArray.add(42); // nope!! Arrays don't have functions
secondArray[10] = 42; // nope!! Off the end of the memory
                      // space you were given
```

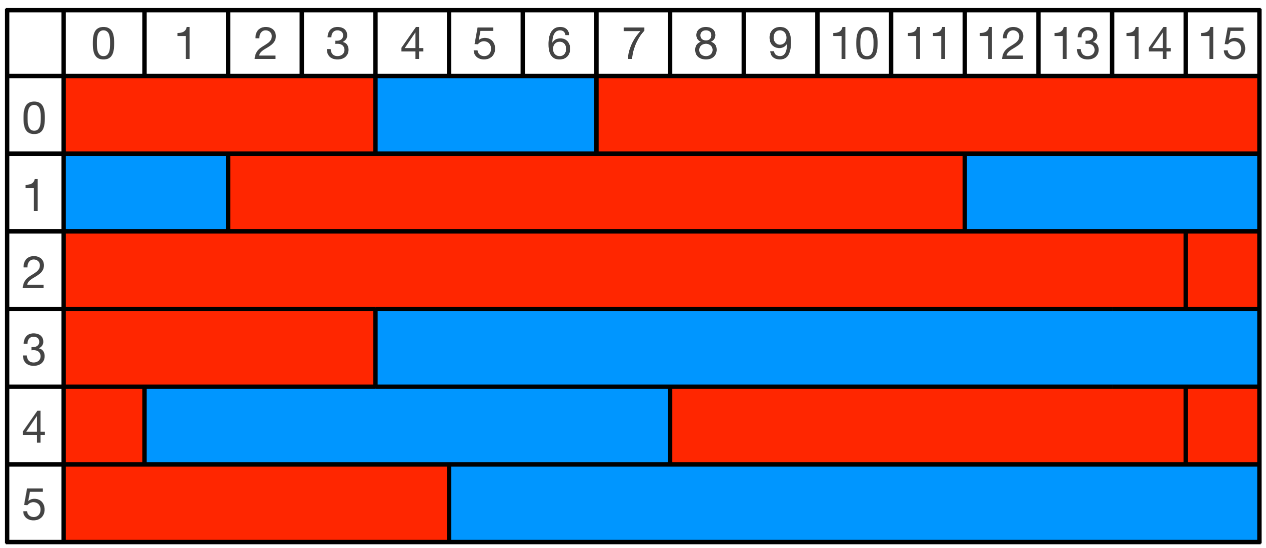
- When using arrays, you have to work with the limitations. We're taking off the training wheels!

Dynamic Memory Allocation: under the hood

- The following statement requests an array of ten integers from the operating system (OS).

```
int *tenInts = new int[10]; // create 10 integers on the heap
```

- The OS looks for enough unallocated memory on the *heap* in a row to give you, then returns a pointer to that location (**red** is used, **blue** is free):

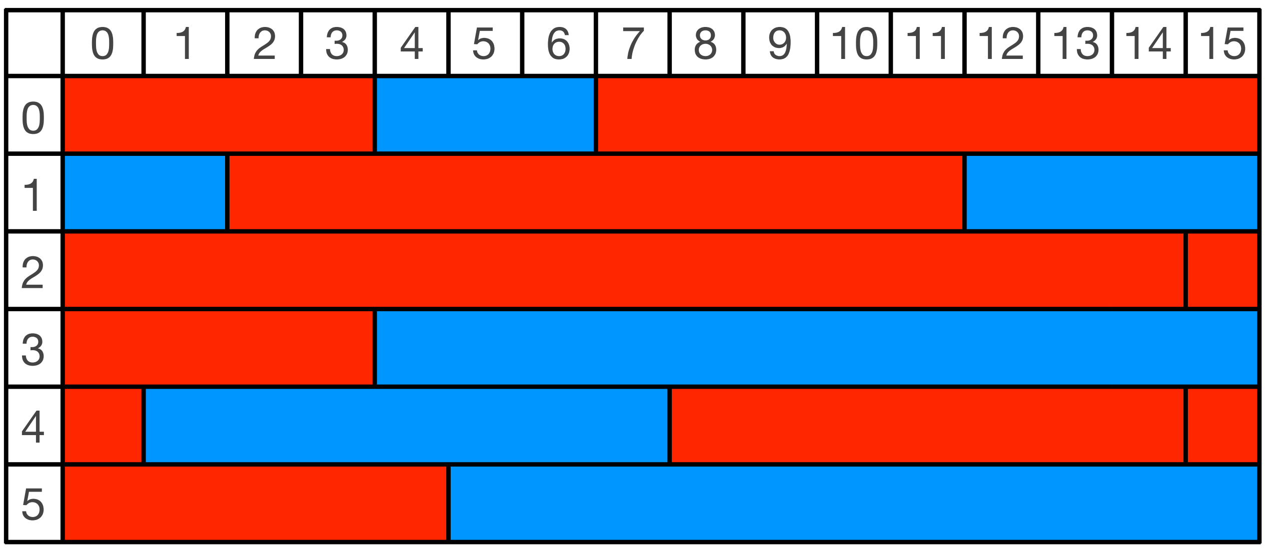


Dynamic Memory Allocation: under the hood

- The following statement requests an array of ten integers from the operating system (OS).

```
int *tenInts = new int[10]; // create 10 integers on the heap
```

- For the above statement, the OS might pick row 3, column 4 for your request.

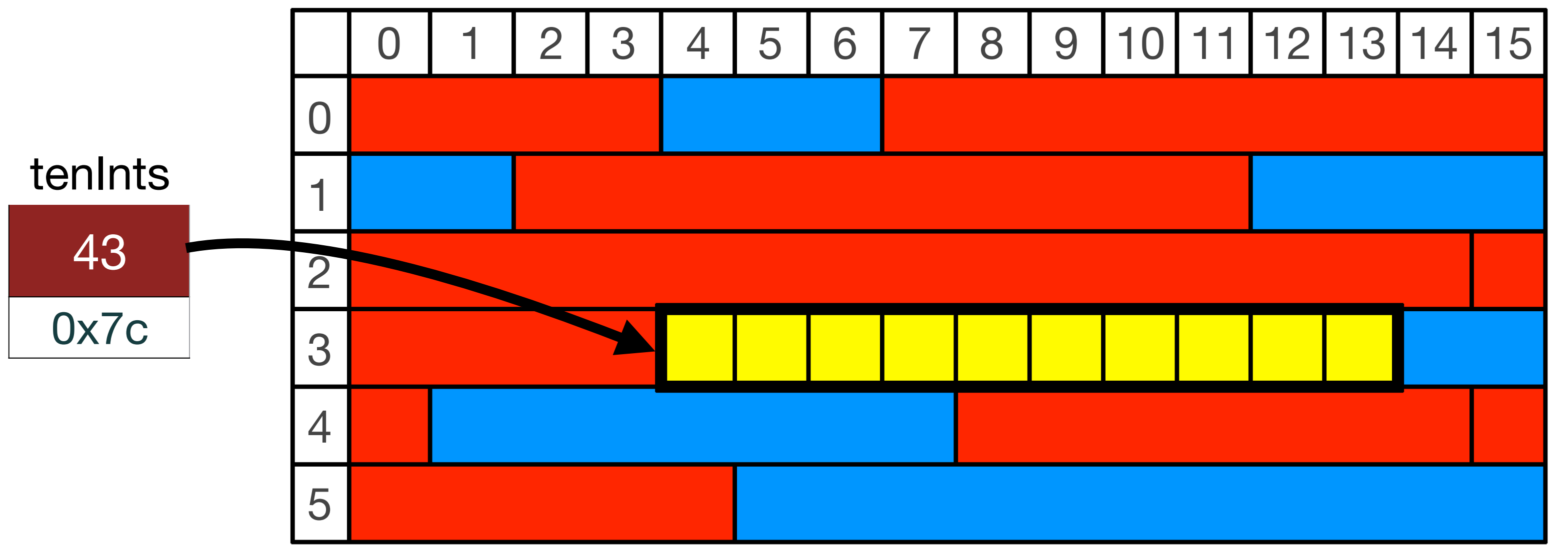


Dynamic Memory Allocation: under the hood

- The following statement requests an array of ten integers from the operating system (OS).

```
int *tenInts = new int[10]; // create 10 integers on the heap
```

- For the above statement, the OS might pick row 3, column 4 for your request.

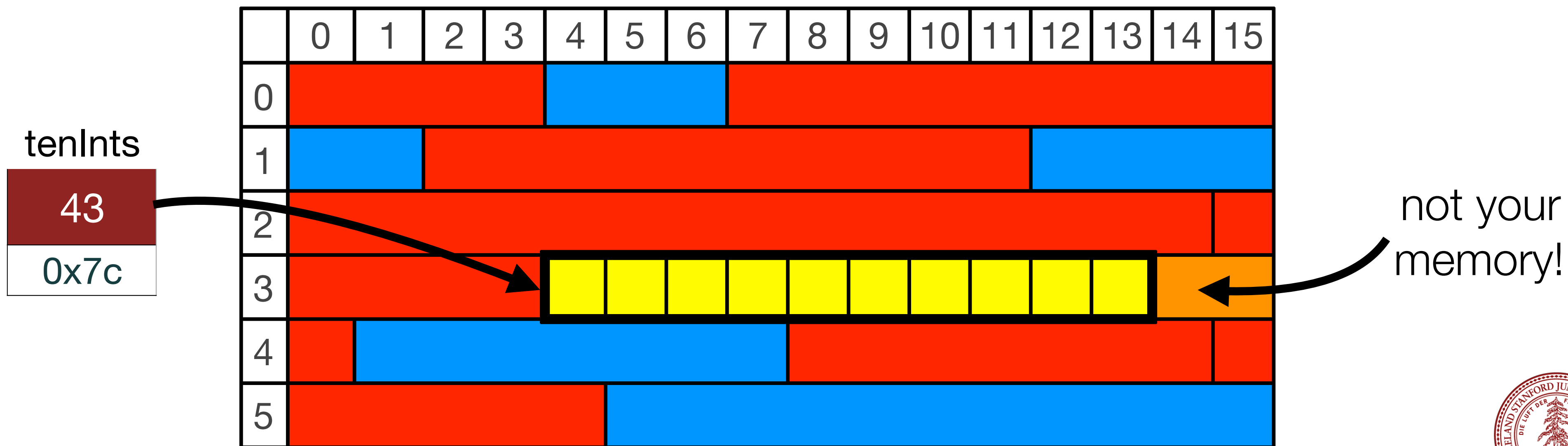


Dynamic Memory Allocation: under the hood

- The following statement requests an array of ten integers from the operating system (OS).

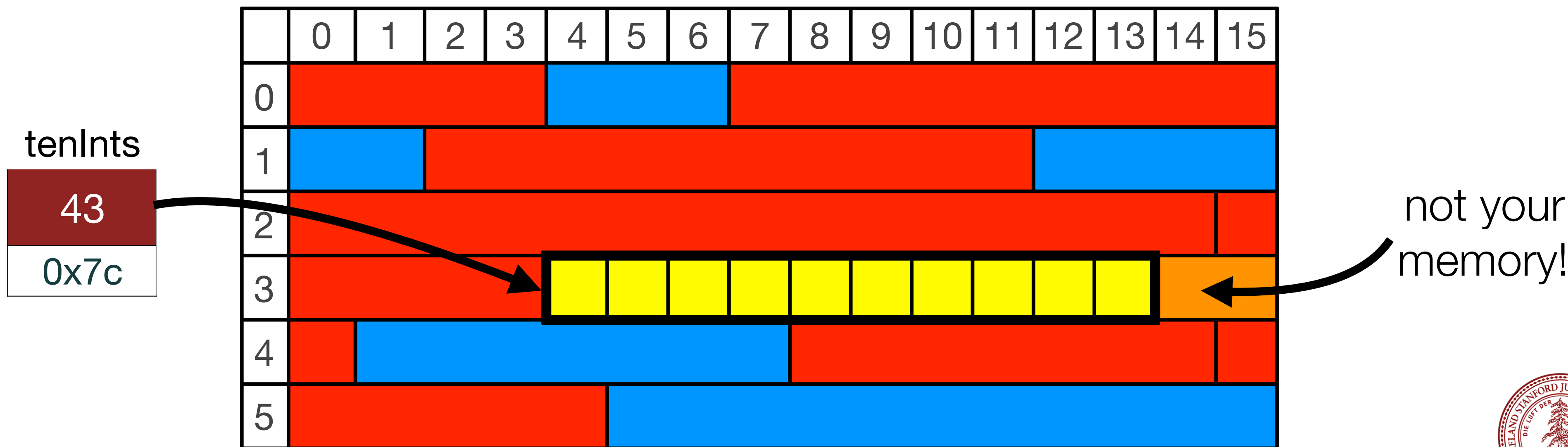
```
int *tenInts = new int[10]; // create 10 integers on the heap
```

- For the above statement, the OS might pick row 3, column 4 for your request.



Dynamic Memory Allocation: under the hood

- What would happen if you *do* try to write a value into a location you don't own?
- Possibilities:
  1. Compiler won't let you.
  2. Crash (seg fault)
  3. Nothing, as no one else is using that area
  4. Headline news for you in the New York Times.

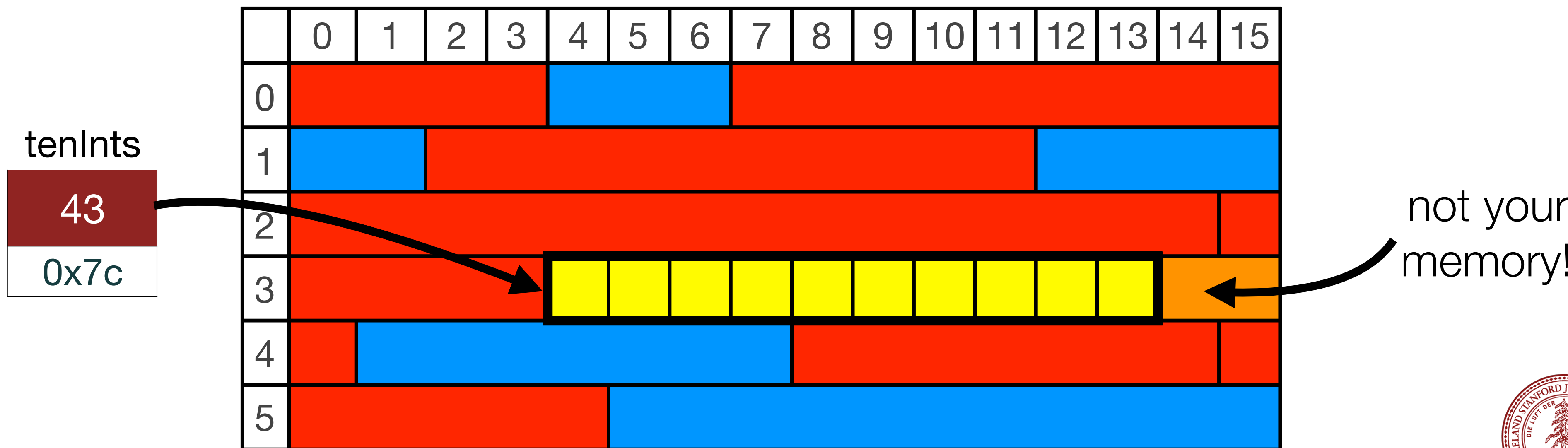


Dynamic Memory Allocation: under the hood

- What would happen if you *do* try to write a value into a location you don't own?
- Possibilities:

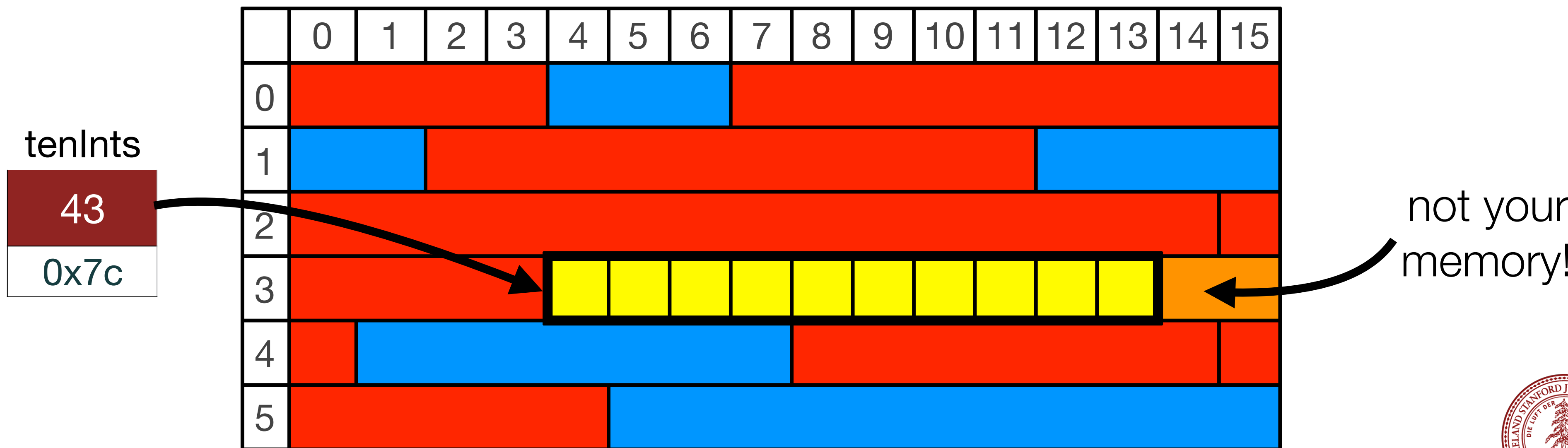
  1. Compiler won't let you.
  2. Crash (seg fault)
  3. Nothing, as no one else is using that area
  4. Headline news for you in the New York Times.

Sadly, the compiler can't tell. You're on your own!



Dynamic Memory Allocation: under the hood

- What would happen if you *do* try to write a value into a location you don't own?
- Possibilities:
  1. Compiler won't let you.
  2. Crash (seg fault) Maybe. The OS can say "I don't think so!" but it isn't guaranteed.
  3. Nothing, as no one else is using that area
  4. Headline news for you in the New York Times.

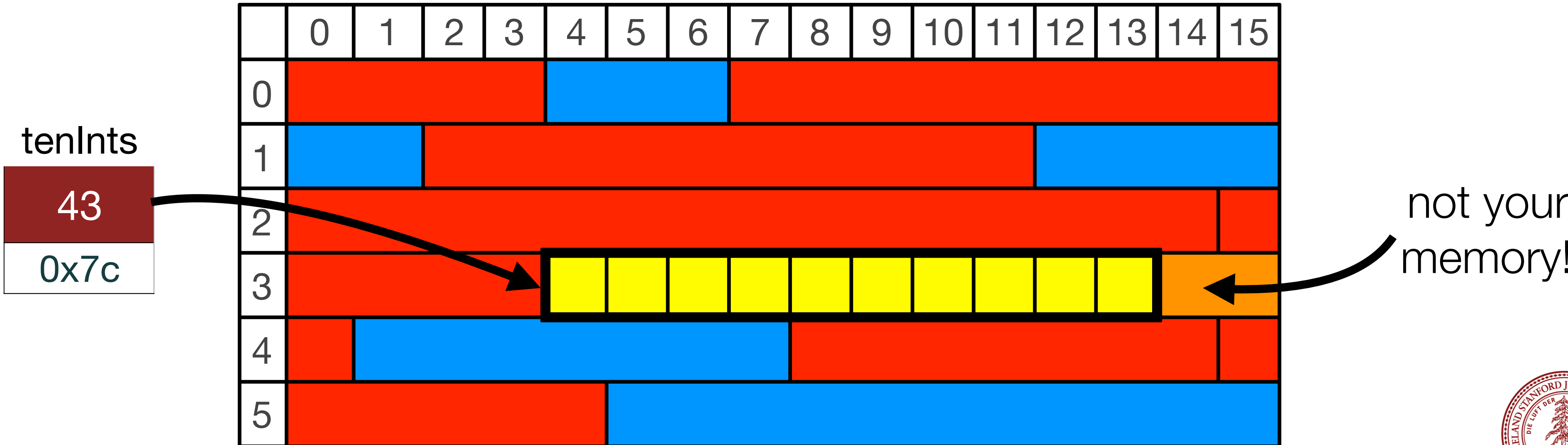


Dynamic Memory Allocation: under the hood

- What would happen if you *do* try to write a value into a location you don't own?
- Possibilities:

1. Compiler won't let you.
2. Crash (seg fault)
3. Nothing, as no one else is using that area
4. Headline news for you in the New York Times.

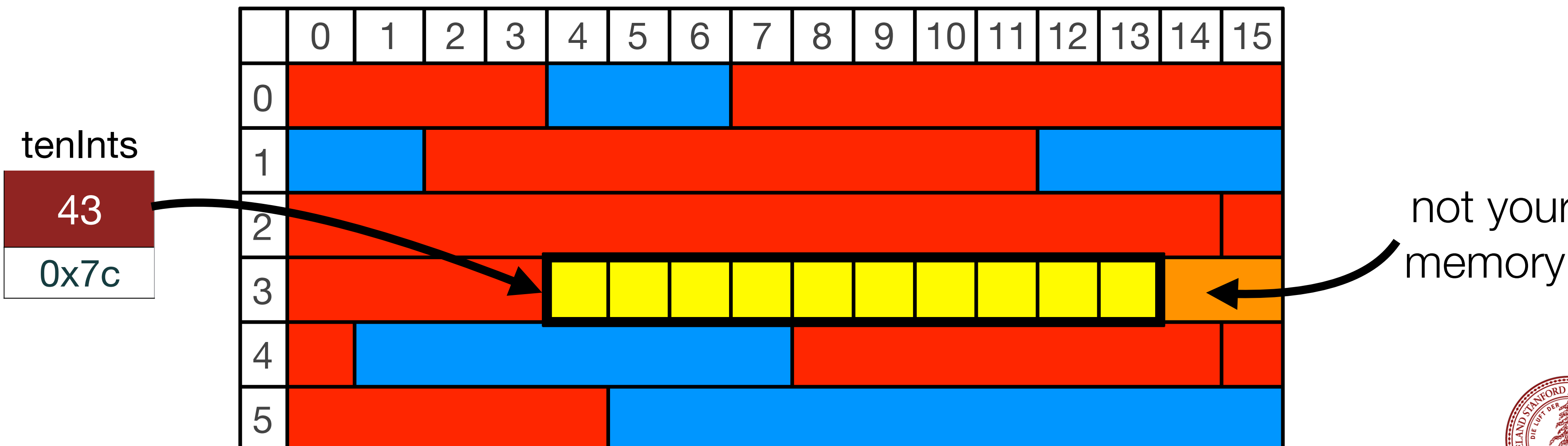
Maybe. The OS might be okay with it, for now...it isn't guaranteed.



Dynamic Memory Allocation: under the hood

- What would happen if you *do* try to write a value into a location you don't own?
- Possibilities:
  1. Compiler won't let you.
  2. Crash (seg fault)
  3. Nothing, as no one else is using that area
  4. Headline news for you in the New York Times.

...?



Buffer Overflows

"All the News That's Fit to Print"

The New York Times

Late Edition

New York: Today, partly sunny, milder. High 50-64. Tonight, mostly cloudy. Low 48-54. Tomorrow, cloudy, windy, rain developing. High 57-62. Yesterday: High 58, low 41. Details, page D16.

VOL. CXXXVIII . . . No. 47,679 — Copyright © 1988 The New York Times — NEW YORK, FRIDAY, NOVEMBER 4, 1988 — 50 cents beyond 75 miles from New York City, except on Long Island. — 35 CENTS

Gov. Michael S. Dukakis having his picture taken by a 10-year-old fan at a town meeting in Fairless Hills, Pa., during a tour of the Northeast in which he emphasized the drug problem. Page A19. Vice President Bush addressed supporters a rally in Columbus, Ohio. Less than a week after Mr. Dukakis acknowledged being a liberal, Mr. Bush said yesterday that "this election is not about labels." Page A18.

'Virus' in Military Computers Disrupts Systems Nationwide

By JOHN MARKOFF

In an intrusion that raises questions about the vulnerability of the nation's computers, a Department of Defense network has been disrupted since Wednesday by a rapidly spreading "virus" program apparently introduced by a computer science student.

The program reproduced itself through the computer network, making hundreds of copies in each machine it reached, effectively clogging systems linking thousands of military, corporate and university computers around the nation and preventing them from doing additional work. The virus is thought not to have destroyed any files.

By late yesterday afternoon computer experts were calling the virus the largest assault ever on the nation's computers.

'The Big Issue'

"The big issue is that a relatively benign software program can virtually bring our computing community to its knees and keep it there for some time," said Chuck Cole, deputy computer security manager at Lawrence Livermore Laboratory in Livermore, Calif., one of the sites affected by the intrusion. "The cost is going to be staggering."

Clifford Stoll, a computer security expert at Harvard University, added: "There is not one system manager who is not tearing his hair out. It's causing enormous headaches."

The affected computers carry a tremendous variety of business and research information among military officials, researchers and corporations.

While some sensitive military data are involved, the computers handling the nation's most sensitive secret information, like that on the control of nuclear weapons, are thought not to have been touched by the virus.

Parallel to Biological Virus

Computer viruses are so named because they parallel in the computer world the behavior of biological viruses. A virus is a program, or a set of instructions to a computer, that is either planted on a floppy disk meant to be used with the computer or introduced when the computer is communicating over telephone lines or data networks with other computers.

The programs can copy themselves into the computer's master software, or operating system, usually without calling any attention to themselves. From there, the program can be passed to additional computers.

Depending upon the intent of the software's creator, the program might cause a provocative but otherwise harmless message to appear on the computer's screen. Or it could systematically destroy data in the computer's memory. In this case, the virus program did nothing more than reproduce itself rapidly.

The program was apparently a result of an experiment, which

Continued on Page A21, Column 2

PENTAGON REPORTS IMPROPER CHARGES FOR CONSULTANTS

CONTRACTORS CRITICIZED

Inquiry Shows Routine Billing of Government by Industry on Fees, Some Dubious

By JOHN H. CUSHMAN Jr.

Special to The New York Times

WASHINGTON, Nov. 3 — A Pentagon investigation has found that the nation's largest military contractors routinely charge the Defense Department for hundreds of millions of dollars paid to consultants, often without justification.

The report of the investigation said that neither the military's current rules nor the contractors' own policies are adequate to assure that the Government does not improperly pay for privately arranged consulting work. Senior Defense Department officials said the Pentagon was proposing changes to correct the flaws.

While it is not improper for military contractors to use consultants in performing work for the Pentagon, the work must directly benefit the military if it is to be paid for by the Defense Department. Often, Pentagon investigators discovered, this test is not met.

Broader Look at Consultants

The Justice Department's continuing criminal investigation has focused attention on consultants and their role in the designing and selling of weapons, and the Defense Department has been criticized for using consultants too freely. Now the Pentagon's own inves-

Registration Off Since 1984 Vote

There has been a pronounced decline in the percentage of eligible Americans who are registered to vote, a research group reports.

Nationally, the percentage of eligible Americans who are registered is estimated to be 70.9 percent, down 2.2 points from the 1984 level.

The group's study concluded that in many of the 30 states where final figures are available the decline was among

Buffer Overflows

- In 1988, a computer "worm" written by Cornell graduate student Robert Morris, Jr. proliferated through government and university computers, bringing down the nascent Internet.
- The worm took advantage of a "buffer overflow" in a program, by writing code into a location that was outside the area that the program was given.


"All the News That's Fit to Print"

The New York Times

Late Edition

New York: Today, partly sunny, milder. High 50-54. Tonight, mostly cloudy. Low 48-54. Tomorrow, cloudy, windy, rain developing. High 57-62. Yesterday: High 58, low 41. Details, page D16.

VOL. CXXXVIII . . . No. 47,679 — NEW YORK, FRIDAY, NOVEMBER 4, 1988 — 35 CENTS

'Virus' in Military Computers Disrupts Systems Nationwide

By JOHN MARKOFF

In an intrusion that raises questions about the vulnerability of the nation's computers, a Department of Defense network has been disrupted since Wednesday by a rapidly spreading "virus" program apparently introduced by a computer science student.

The program reproduced itself through the computer network, making hundreds of copies in each machine it reached, effectively clogging systems linking thousands of military, corporate and university computers around the nation and preventing them from doing additional work. The virus is thought not to have destroyed any files.

By late yesterday afternoon computer experts were calling the virus the largest assault ever on the nation's computers.

'The Big Issue'

"The big issue is that a relatively benign software program can virtually bring our computing community to its knees and keep it there for some time," said Chuck Cole, deputy computer security manager at Lawrence Livermore Laboratory in Livermore, Calif., one of the sites affected by the intrusion. "The cost is going to be staggering."

Clifford Stoll, a computer security expert at Harvard University, added: "There is not one system manager who is not tearing his hair out. It's causing enormous headaches."

The affected computers carry a tremendous variety of business and research information among military officials, researchers and corporations.

While some sensitive military data are involved, the computers handling the nation's most sensitive secret information, like that on the control of nuclear weapons, are thought not to have been touched by the virus.

Parallel to Biological Virus

Computer viruses are so named because they parallel in the computer world the behavior of biological viruses. A virus is a program, or a set of instructions to a computer, that is either planted on a floppy disk meant to be used with the computer or introduced when the computer is communicating over telephone lines or data networks with other computers.

The programs can copy themselves into the computer's master software, or operating system, usually without calling any attention to themselves. From there, the program can be passed to additional computers.

Depending upon the intent of the software's creator, the program might cause a provocative but otherwise harmless message to appear on the computer's screen. Or it could systematically destroy data in the computer's memory. In this case, the virus program did nothing more than reproduce itself rapidly.

The program was apparently a result of an experiment, which

Continued on Page A21, Column 2

Gov. Michael S. Dukakis having his picture taken by a 10-year-old fan at a town meeting in Fairless Hills, Pa., during a tour of the Northeast in which he emphasized the drug problem. Page A10. Vice President Bush addressed supporters a rally in Columbus, Ohio. Less than a week after Mr. Dukakis acknowledged being a liberal, Mr. Bush said yesterday that "this election is not about labels." Page A18.

Registration Off Since 1984 Vote

There has been a pronounced decline in the percentage of eligible Americans who are registered to vote, a research group reports.

Nationally, the percentage of eligible Americans who are registered is estimated to be 70.9 percent, down 2.2 points from the 1984 level.

The group's study concluded that in many of the 30 states where final figures are available the decline was among

PENTAGON REPORTS IMPROPER CHARGES FOR CONSULTANTS

CONTRACTORS CRITICIZED

Inquiry Shows Routine Billing of Government by Industry on Fees, Some Dubious

By JOHN H. CUSHMAN Jr.

Special to The New York Times

WASHINGTON, Nov. 3 — A Pentagon investigation has found that the nation's largest military contractors routinely charge the Defense Department for hundreds of millions of dollars paid to consultants, often without justification.

The report of the investigation said that neither the military's current rules nor the contractors' own policies are adequate to assure that the Government does not improperly pay for privately arranged consulting work. Senior Defense Department officials said the Pentagon was proposing changes to correct the flaws.

While it is not improper for military contractors to use consultants in performing work for the Pentagon, the work must directly benefit the military if it is to be paid for by the Defense Department. Often, Pentagon investigators discovered, this test is not met.

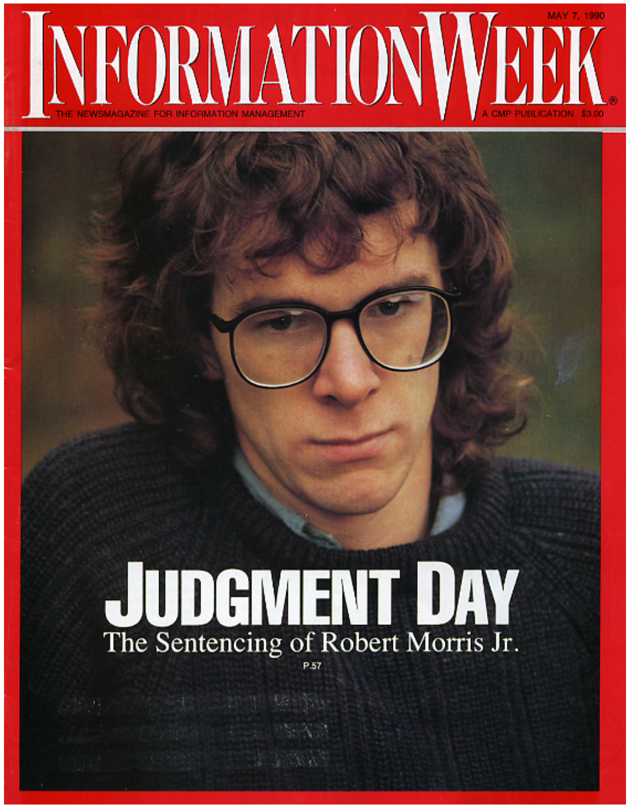
Broader Look at Consultants

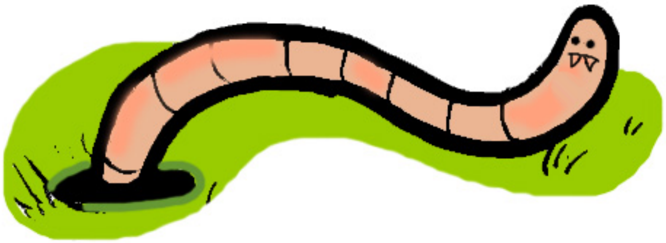
The Justice Department's continuing criminal investigation has focused attention on consultants and their role in the designing and selling of weapons, and the Defense Department has been criticized for using consultants too freely. Now the Pentagon's own inves-

- The worm tricked the program into running its code, and was able to work its way through the network to other computers.
- The worm had a bug that made it eat up all of the computer's memory, thereby crashing the systems, one by one.

Buffer Overflows

- Robert Morris, Jr. became the first person in the U.S. convicted under the Computer Fraud and Abuse Act, and was fined, performed community service and served a three-year probation.
- He claimed that he was trying to demonstrate computer security faults, but the court did not believe him.
- He did bounce back: now he is a professor of computer science at MIT, and he co-founded the start-up incubator, Y-Combinator.





Dynamic Memory Allocation: delete

- The memory you request is yours until the end of the program, if you need it that long.
- You can pass around the pointer you get back as much as you'd like, and you have access to that memory through that pointer in any function you pass the pointer to.
- But, what if you are done using that memory? Let's say you create an array of 10 ints, use them for some task, and then are done with the memory?
- In this case, you *delete* the memory, giving it back to the Operating System:

```
int *tenInts = new int[10]; // create 10 integers on the heap
for (int i=0; i < 10; i++) {
    tenInts[i] = randomInteger(1,1000);
}
someFunction(tenInts);
// done using tenInts
delete [] tenInts; // the [] is necessary for an array
```

Dynamic Memory Allocation: delete

- **`delete`** is sometimes confusing. Take a look at the following function:

```
void arrayFun(int *origArray, int length) {
    // allocate space for a new array
    int *multiple = new int[length];

    for (int i=0; i < length; i++) {
        multiple[i] = origArray[i] * 2; // double each value
    }

    printArray(multiple, length); // prints each value doubled

    delete [] multiple; // give back the memory

    multiple = new int[length * 2]; // now twice as many
    for (int i=0; i < length; i++) {
        multiple[i*2] = origArray[i] * 2; // double each value
        multiple[i*2+1] = origArray[i] * 3; // triple the value
    }

    printArray(multiple, length * 2);

    delete [] multiple; // clean up
}
```

Dynamic Memory Allocation: delete

- **`delete`** is sometimes confusing. Take a look at the following function:

```
void arrayFun(int *origArray, int length) {
    // allocate space for a new array
    int *multiple = new int[length];

    for (int i=0; i < length; i++) {
        multiple[i] = origArray[i] * 2; // double each value
    }

    printArray(multiple, length); // prints each value doubled

    delete [] multiple; // give back the memory

    multiple = new int[length * 2]; // now twice as many
    for (int i=0; i < length; i++) {
        multiple[i*2] = origArray[i] * 2; // double each value
        multiple[i*2+1] = origArray[i] * 3; // triple the value
    }

    printArray(multiple, length * 2);

    delete [] multiple; // clean up
}
```

- First: notice that we delete multiple, and then use it again!
- Is that allowed??

Dynamic Memory Allocation: delete

- **`delete`** is sometimes confusing. Take a look at the following function:

```
void arrayFun(int *origArray, int length) {
    // allocate space for a new array
    int *multiple = new int[length];

    for (int i=0; i < length; i++) {
        multiple[i] = origArray[i] * 2; // double each value
    }

    printArray(multiple, length); // prints each value doubled

    delete [] multiple; // give back the memory

    multiple = new int[length * 2]; // now twice as many
    for (int i=0; i < length; i++) {
        multiple[i*2] = origArray[i] * 2; // double each value
        multiple[i*2+1] = origArray[i] * 3; // triple the value
    }

    printArray(multiple, length * 2);

    delete [] multiple; // clean up
}
```

- First: notice that we delete multiple, and then use it again!
- Is that allowed??
    - It is! **`delete`** *does not delete any variables! Instead, it follows the pointer and returns the memory to the OS!*
    - However, you are not allowed to use the memory after you have **`delete`**d it.
    - This does not preclude you from re-using the pointer itself.

Dynamic Memory Allocation: delete

- What does this print out, by the way for an **`origArray = {1, 5, 7}`**?

```
void arrayFun(int *origArray, int length) {
    // allocate space for a new array
    int *multiple = new int[length];

    for (int i=0; i < length; i++) {
        multiple[i] = origArray[i] * 2; // double each value
    }

    printArray(multiple, length); // prints each value doubled

    delete [] multiple; // give back the memory

    multiple = new int[length * 2]; // now twice as many
    for (int i=0; i < length; i++) {
        multiple[i*2] = origArray[i] * 2; // double each value
        multiple[i*2+1] = origArray[i] * 3; // triple the value
    }

    printArray(multiple, length * 2);

    delete [] multiple; // clean up
}
```

```
void printArray(int *array,
                int length) {
    cout << "[";
    for (int i=0; i < length; i++) {
        cout << array[i];
        if (i < length-1) {
            cout << ", ";
        }
    }
    cout << "]" << endl;
}
```

Output:

[2, 10, 14]
[2, 3, 10, 15, 14, 21]

Dynamic Memory Allocation: under the hood

- The memory you request is yours until the end of the program, if you need it that long.
- You can pass around the pointer you get back as much as you'd like, and you have access to that memory through that pointer in any function you pass the pointer to.

- Without knowing it, you have been using dynamic memory all along, through the use of the standard and Stanford library classes. The string, Vector, Map, Set, Stack, Queue, etc., all use dynamic memory to give you the data structures we have used for all our programs.

Thought experiment: the scary world without dynamic memory

- What if (horror!) we took away the Stanford library and asked you to write a Microsoft Word clone. Maybe you would start with something like this (although you'd probably make a Page class, instead):

```
struct Page {
    string text;
    double leftM, rightM, topM, bottomM; // margins
    string header, footer;
    int textColor;
};
```

- How many pages should we allow the user of Stanford Word?

Thought experiment: the scary world without dynamic memory

- What if (horror!) we took away the Stanford library and asked you to write a Microsoft Word clone. Maybe you would start with something like this (although you'd probably make a Page class, instead):

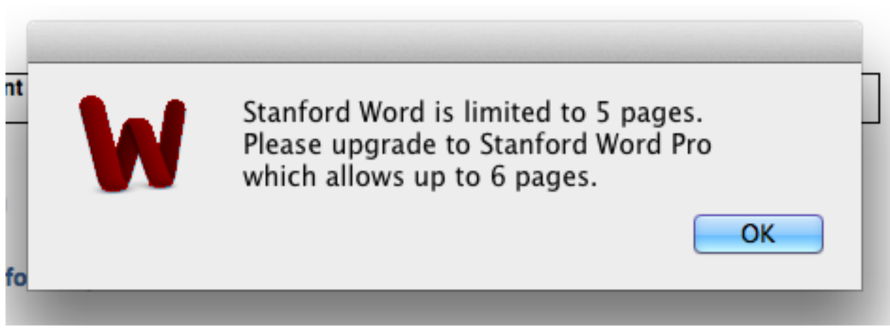
```
struct Page {
    string text;
    double leftM, rightM, topM, bottomM; // margins
    string header, footer;
    int textColor;
};
```

- How many pages should we allow the user of Stanford Word? 5?

```
int main() {
    Page pages[5]; // array of 5 pages
}
```

Thought experiment: the scary world without dynamic memory

- People probably wouldn't buy your program if you limited them to five pages.

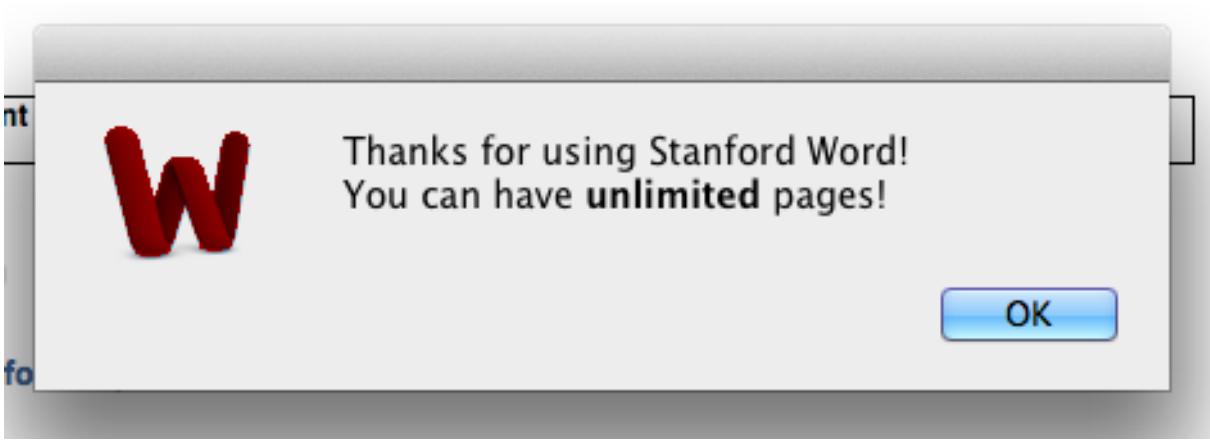


- Okay, let's make it bigger. How big? 6 pages? 100 pages? 1,000,000 pages?
- This is a no-win battle.
  - Too small, and your user might be unhappy.
  - Too big? Waste of memory! Your program would hog memory if you did the following:

```
int main() {
    Page pages[1000000]; // array of a million pages
}
```

Next Time: Building a Vector class with arrays

- In the two lectures, we will discuss how the Vector is built, using dynamic memory.
- We will need to keep track of all the details ourselves:
  - How much space we have allocated for the Vector
  - How many items are in the Vector
  - How to add / remove / insert into the Vector
  - How to *expand* the Vector



Recap

- Dynamic Memory Allocation:
  - **`new`**: used to request heap memory that lasts for the rest of your program, or until you don't need it anymore.
  - **`delete`**: used to return memory to the operating system.
  - If you use **`new`** to request memory, you should **`delete`** it somewhere in your program.
  - You are **not allowed** to use memory that has been **`delete`**d.
  - deleting memory does not somehow "delete" the pointer variable -- it goes to the location in memory pointed to, and tells the operating system that we are done with it.

References and Advanced Reading

- **References:**
  - **`new`** and **`delete`**: https://www.tutorialspoint.com/cplusplus/cpp_dynamic_memory.htm
  - Video on dynamic memory allocation: https://www.youtube.com/watch?v=OrDjGp_y1H4

- **Advanced Reading:**
  - Fun video on pointers: https://www.youtube.com/watch?v=B7lVHq-cgeU
  - Morris Worm: https://en.wikipedia.org/wiki/Morris_worm
  - Buffer Overflow vulnerabilities: https://en.wikipedia.org/wiki/Buffer_overflow

Chris's Favorite YouTube Channels

- Numberphile: https://www.youtube.com/user/numberphile
  - Cool videos about math, mathematical games and puzzles, number theory, etc.

- Computerphile: https://www.youtube.com/channel/UC9-y-6csu5WGm29I7JiwpnA
  - Sister site (produced by the same folks) as Numberphile, focusing on computers

- standupmaths: https://www.youtube.com/channel/UCSju5G2aFaWMqn-_0YBtq5A
  - Numberphile spin-off from one of its contributors

- EEVBlog: https://www.youtube.com/channel/UCr-cm90DwFJC0W3f9jBs5jA
  - If you are interested in electronics, this is the place to go. Great tutorials, commentary, etc., by a down-to-earth Australian. "Don't turn it on, take it apaht!"

- bigclivedotcom: https://www.youtube.com/channel/UCtM5z2gkrGRuWd0JQMx76qA
  - Scottish version of EEVBlog, with detailed under-the-hood electronics

- AvE: https://www.youtube.com/channel/UChWv6Pn_zP0rI6lgGt3MyfA
  - Irreverent Canadian who demonstrates the mechanical engineering end of the spectrum

- Kurzgesagt — In a Nutshell: https://www.youtube.com/user/Kurzgesagt
  - Short videos on cool stuff

- VSauce: https://www.youtube.com/user/Vsauce
  - Science, math, philosophy, etc.

Extra Slides (will cover next time)

Dynamic Memory Allocation: your responsibilities

- With great power comes great responsibility
- You have a responsibility when using dynamic memory allocation to **`delete`** anything you have requested via **`new`**.
- This is the contract you make with the operating system: if you're done with the memory, you should return it. The OS will take it back when your program ends, but this wastes memory, and this is called a "memory leak."

```
const int INIT_CAPACITY = 10000000;

class Demo {
public:
    Demo(); // constructor
    string at(int i);
private:
    string *bigArray;
};

Demo::Demo() {
    bigArray = new string[INIT_CAPACITY];
    for (int i=0;i<INIT_CAPACITY;i++) {
        bigArray[i] = "Lalalalalalalalala!";
    }
}
```

```
string Demo::at(int i) {
    return bigArray[i];
}

int main()
{
    for (int i=0;i<10000;i++){
        Demo demo;
        cout << i << ": " << demo.at(1234) << endl;
    }
    return 0;
}
```

This program crashed my entire computer when I ran it. Why?

Dynamic Memory Allocation: your responsibilities

- This program crashed my entire computer when I ran it. Why?
- We're allocating a *ton* of memory, and not deleting it!
- We can fix it by adding a "destructor" -- when the class instance goes out of scope, the destructor is called, cleaning up the memory for us.

```cpp
const int INIT_CAPACITY = 10000000;

class Demo {
public:
    Demo(); // constructor
    ~Demo(); // destructor
    string at(int i);
private:
    string *bigArray;
};

Demo::Demo() {
    bigArray = new string[INIT_CAPACITY];
    for (int i=0;i<INIT_CAPACITY;i++) {
        bigArray[i] = "Lalalalalalalalala!";
    }
}
```

```cpp
Demo::~Demo() {
    delete[] big_array;
}

string Demo::at(int i) {
    return bigArray[i];
}

int main()
{
    for (int i=0;i<10000;i++){
        Demo demo;
        cout << i << ": " << demo.at(1234) << endl;
    }
    return 0;
}
```