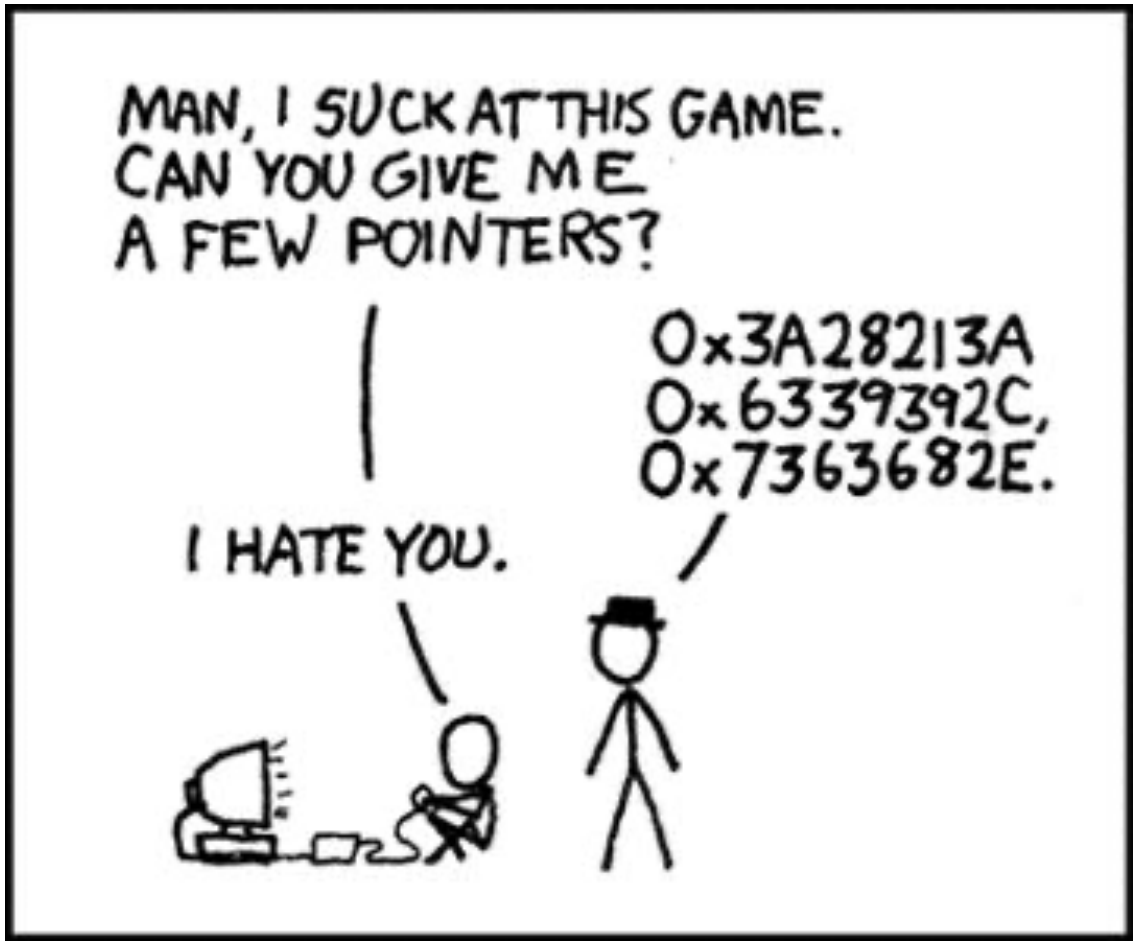
Lecture #3

- Pointers:
  - A Quick Review of Pointers
  - Dynamic Memory Allocation
- Resource Management Part 1:
  - Copy Constructors

If you feel uncomfortable with pointers, then study and become an expert before our next class!

(Yeah right… like you're gonna review on your own)

Pointers


MAN, I SUCK AT THIS GAME.
CAN YOU GIVE ME
A FEW POINTERS?
0x3A28213A
0x6339392C,
0x7363682E.
I HATE YOU.

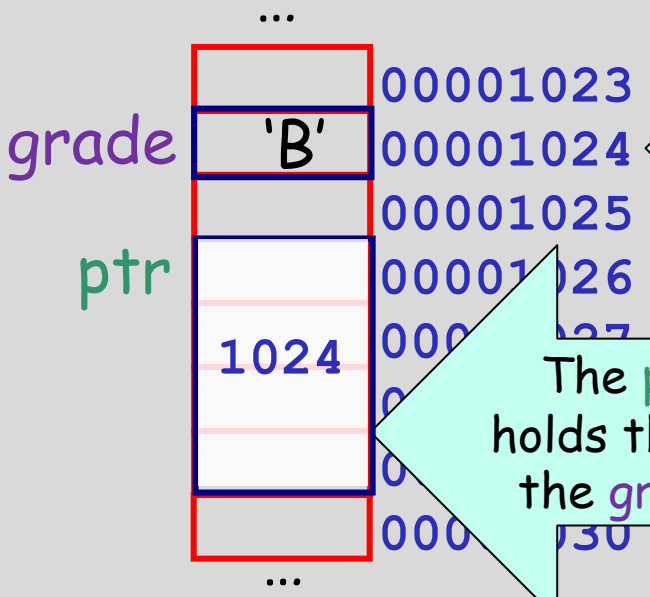
Addresses and Pointers…

What's the big picture?

We use pointers to efficiently access/modify variables defined in other parts of our program.

Just like every house has a street address, every variable has a memory address.

```
void someFunction()
{
    char grade = 'B';
    …

    char *ptr = &grade;
}
```



The grade variable has an address of 1024 in RAM

The ptr variable holds the address of the grade variable

A pointer is simply a variable that holds another variable's address!

Uses:

Pointers are used in all C++ programs to efficiently pass parameters, and to refer to dynamic variables.

Every Variable Has An Address

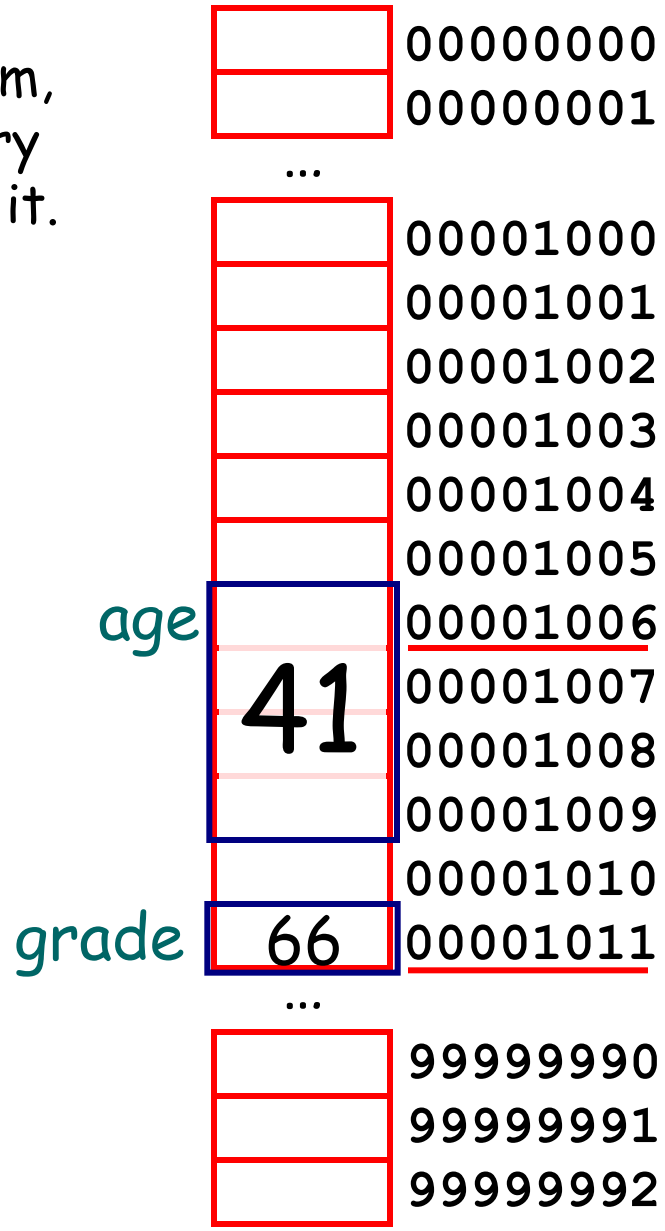
Every time you define a variable in your program, the compiler finds an unused address in memory and reserves one or more bytes there to store it.

Important: The address of a variable is defined to be the *lowest* address in memory where the variable is stored.

So age's address in memory is 1006.

grade's address would be 1011.

```
int main()
{
    int    age = 41;
    char grade = 'B';

    ...

}
```



Getting the Address of a Variable

We can get the address of a variable using C++'s & operator.

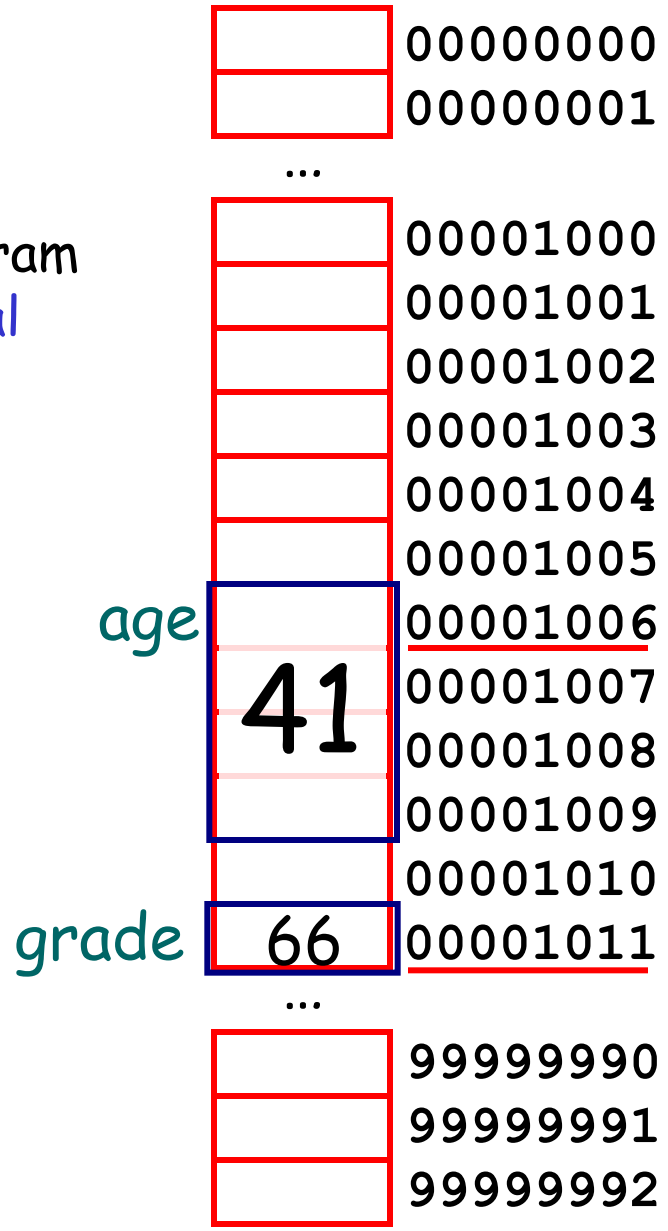
If you place an & before a variable in a program statement, it means "give me the numerical address of the variable."

Output:

age's address: 1006

grade's address: 1011

```
int main()
{
    int    age = 41;
    char grade = 'B';

    cout << "age's address: "<< &age ;
    cout << "grade's address: " << &grade ;
}
```

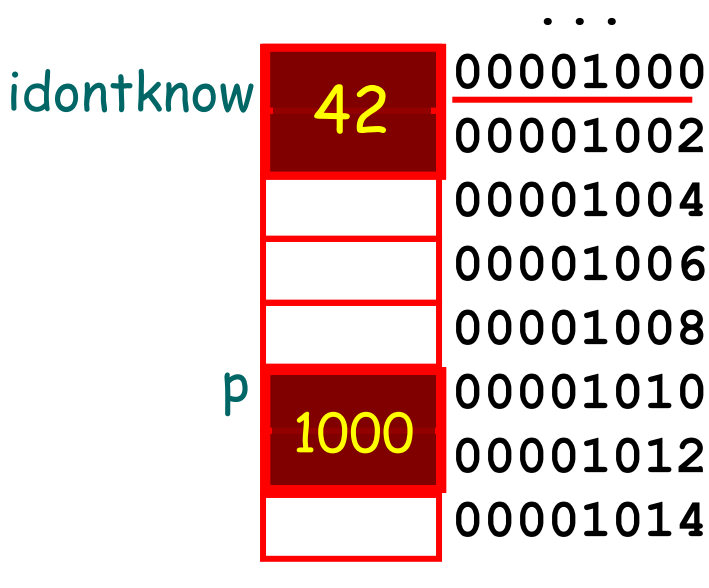


Ok, So What's a Pointer? It's a variable!

A pointer variable is a kind of variable that holds another variable's address instead of a regular value.

- The only difference between a regular variable and a pointer variable is that a pointer variable holds the address of another variable instead of some regular value like 100 or 3.14159 or "barf"
- The way you define a pointer variable is by putting an asterisk * in front of the variable name when you define it, e.g. "int *p;" or "string *s;"
- To understand the type of a pointer variable, simply read your declaration from right to left…
  int *p; → "p is a pointer to an integer variable"
  string *s; → "s is a pointer to a string variable"
  Nerd **n; → "n is a pointer, to a pointer, to a Nerd variable."
- If a pointer variable p holds the address of the idontknow variable, then we can say:
  "p points to idontknow" or "p holds a value of X (like 1000) which is idontknow's address"

```
void foo()
{
   int idontknow;

   idontknow = 42;

   int *p;

   p = &idontknow;
}
```



What do I do with Pointers?

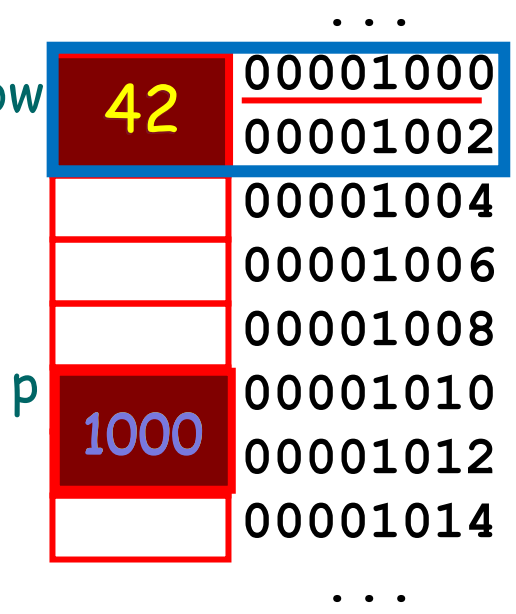
- Question: So I have a pointer variable that points to another variable… now what?
- Answer: You can use your pointer and the star operator to read/write the other variable.
- Line #1 sticks the address of the idontknow variable (1000) into variable p using the & operator to obtain the address of idontknow. So now p points at the idontknow variable.
- Line #2 basically says: "Get the address value stored in the p variable (which is 1000). Then go to that address in memory… and give me the value stored there (which is 42)."

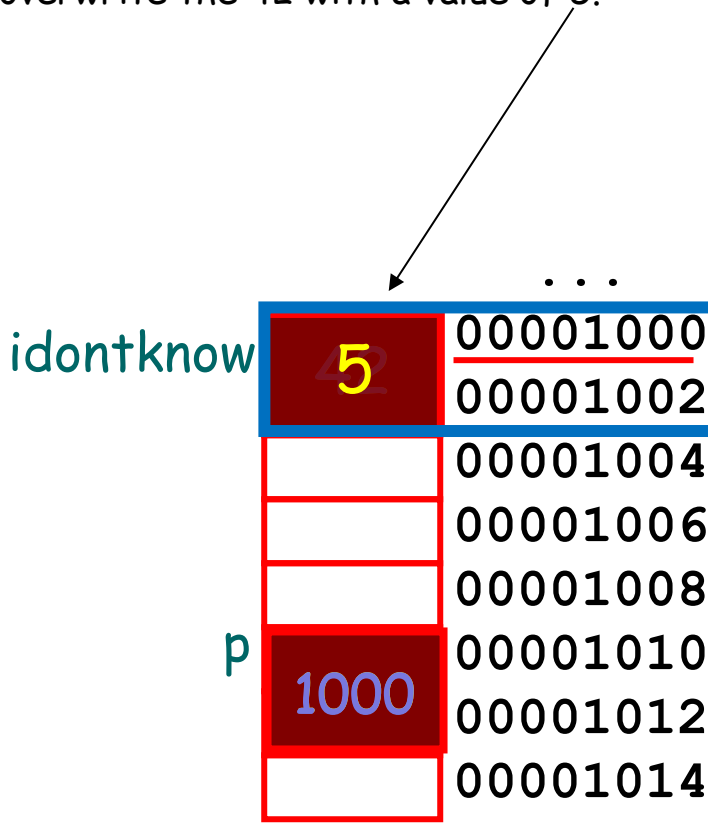
cout << *p → cout << *1000 → cout << 42

- Line #3 basically says: ""Get the address value stored in the p variable (which is 1000). Then go to that address in memory. Then store a value of 5 there." This would overwrite the 42 with a value of 5.

*p = 5 → *1000 = 5

```
void foo()
{
    int idontknow;
    idontknow = 42;

    int *p;
    p = &idontknow; // #1

    cout << *p;  // #2
    *p     = 5; // #3
}
```





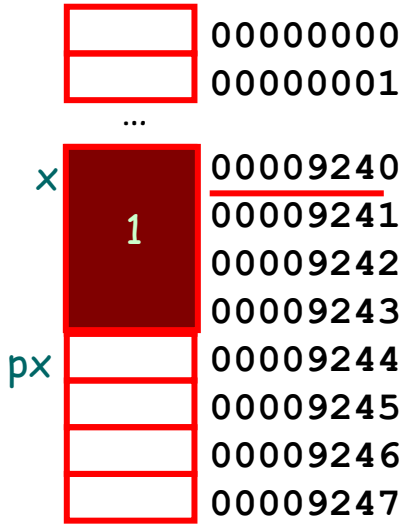
Another Pointer Example

```
void set(int *px)
{
    *px = 5; // #3
}

int main()
{
    int x = 1; // #1

    set(&x);    // #2
    cout << x; // prints 5
}
```

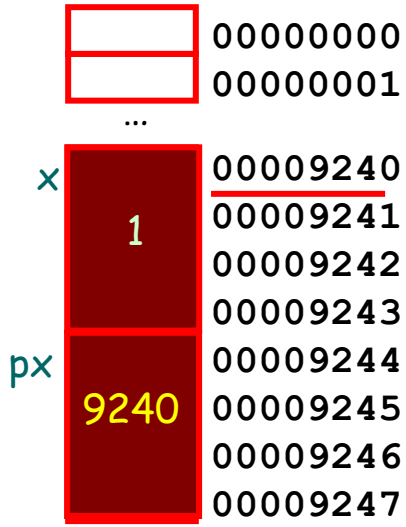
Let's use pointers to modify a variable inside of another function.

Cool – that works! We can use pointers to modify variables from other functions!

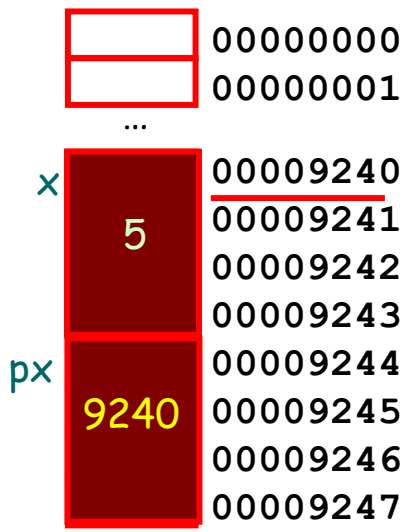
After line #1, we've initialized the value of the x variable to 1.



On line #2, we get the address of variable x, which is 9240 and pass it to the set() function's px parameter.



On line #3, we store a value of 5 where px points to (in location 9240), overwriting the value of 1 there.

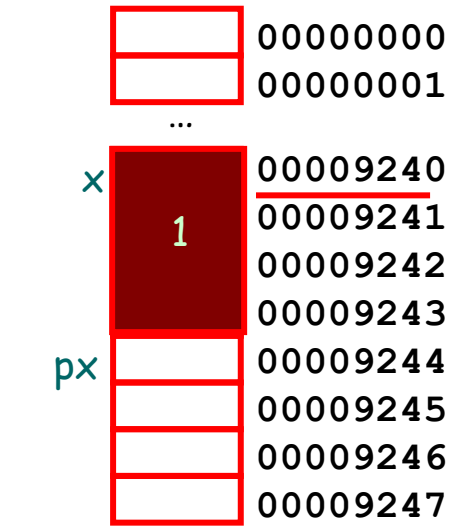


```
void set(int px)
{
   px = 5;    // #3
}

int main()
{
   int x = 1; // #1

   set(x);    // #2
   cout << x; // #4 prints 1
}
```

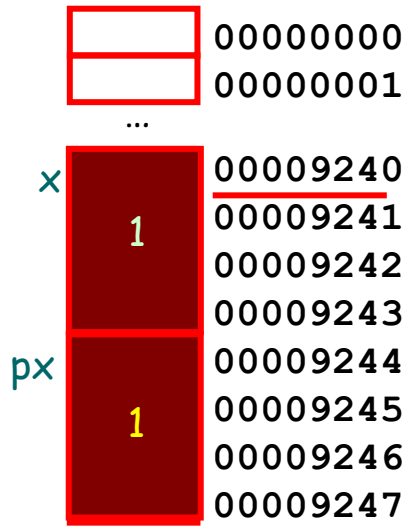
What if We Didn't Use Pointers?

Oh no! We tried to change the value of x in set but it only changed the local variable!

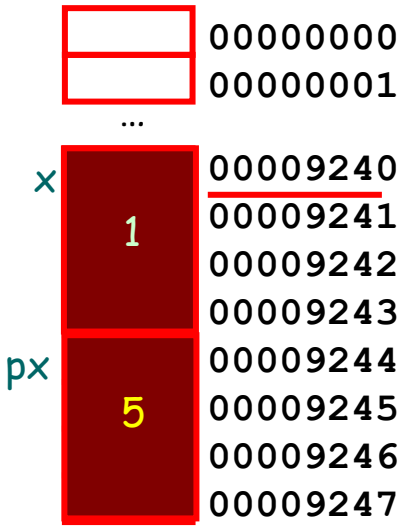
After line #1, we've initialized the value of the x variable to 1.



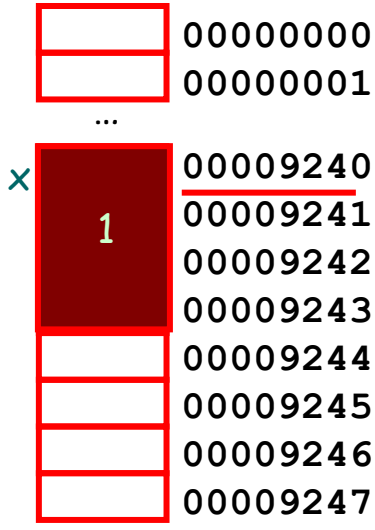
On line #2, we pass the VALUE of variable x, which is 1 to the set() function's px parameter.



On line #3, we store a value of 5 in the px local variable at location 9244. This changes px but has no impact on the original x variable in location 9240!



On line #4, we are back in the main function. The px value has disappeared, and our x value is left unchanged!



Passing by Reference Example

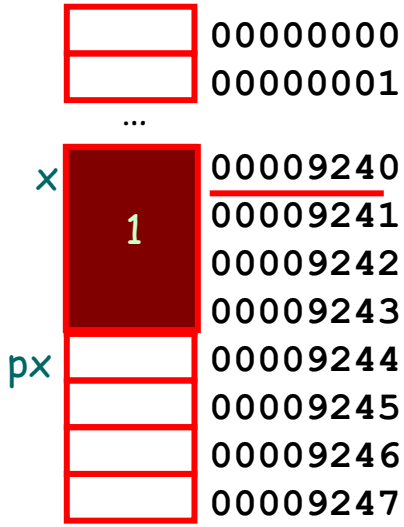
```
void set(int &px)
{
    px = 5; // #3
}

int main()
{
    int x = 1; // #1

    set(x);    // #2
    cout << x; // prints 5
}
```

When you pass a variable by reference to a function, what really happens?

In fact, using a reference is just a simpler notation for passing by a pointer!

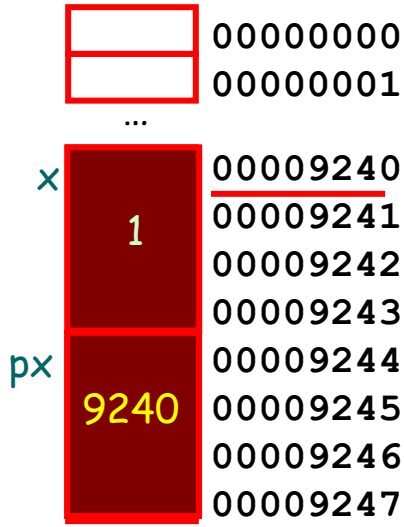
C++ actually uses a pointer under the hood, just like we did a few slides ago.

After line #1, we've initialized the value of the x variable to 1.



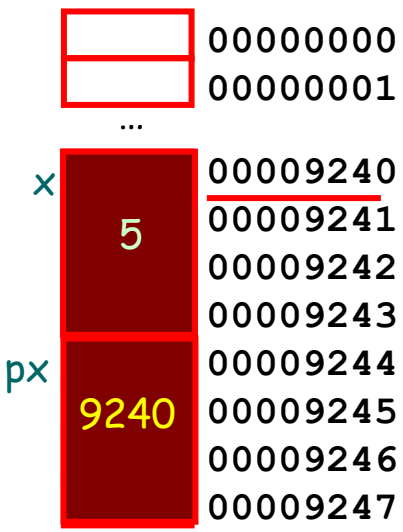
On line #2, it looks like we're just passing the value of x to the set() function. BUT in reality C++ gets the address of variable x, which is 9240 and passes it to the set() function's px parameter.

However this is hidden from us, since px was defined as a reference and not a pointer!



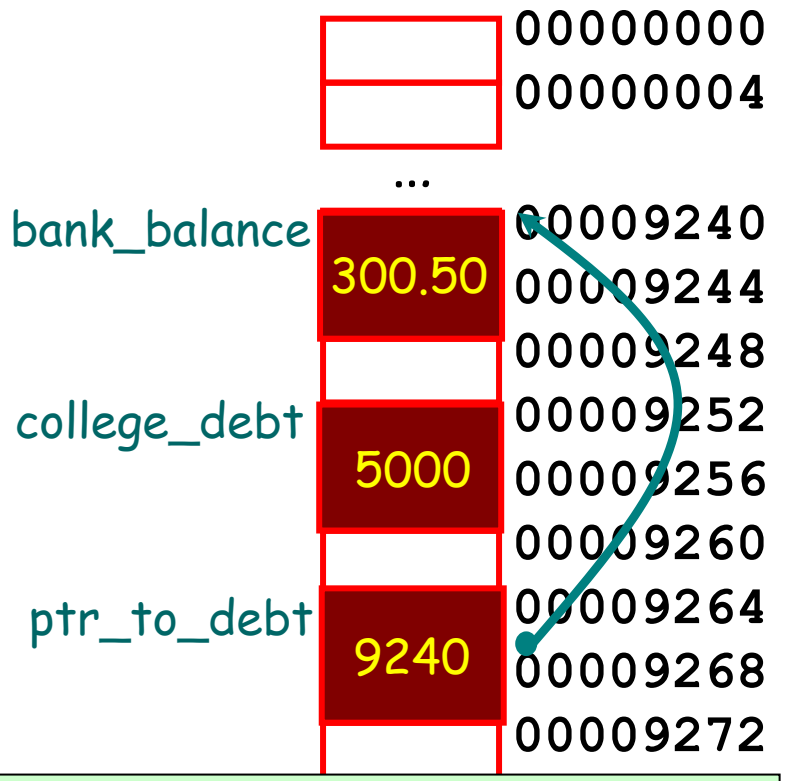
On line #3, we store a value of 5 where px "refers" to (in location 9240), overwriting the value of 1 there.

The fact that px is really a pointer is hidden from us. So when we say
px = 5;
that's exactly the same as saying
*px = 5;
in our first pointer example.



Pointers are Dangerous!

```
int main()
{
   double bank_balance = 325.50; // $$
   double college_debt = 5000; // $$

   double *ptr_to_debt;       // #1

   ...

  *ptr_to_debt = 0;           //  #2
}
```



- We have a very serious bug in the above code.
- The problem is that we define the ptr_to_debt pointer (Line #1) but we never remember to initialize its value before we use the variable on line #2. Because pointers start out with random values, our pointer could hold any address, and essentially point anywhere.
- It might point at our bank balance, it might point at some other random data in our program. We just don't know!
- On line #2, when we set *ptr_to_debt equal to zero, this gets the current value of the pointer variable (shown as 9240 to the right), then goes to that location in memory, and stores a value of zero.
- But that doesn't zero our college_debt variable, as we would have hoped! Instead because the random value in ptr_to_debt just happened to be the address of our bank_balance variable, it was zeroed instead! The horror! We have no money!
- The moral of the story is that you must always explicitly set the address held in a pointer variable or you'll be sorry! ☹
- Pro tip: Always initialize pointers to some other variable's address or to nullptr immediately when you define them! (see example →)
- Why? If you use * on a null pointer, your program will crash immediately and you'll find the bug ASAP!

```
int main()
{
   double bank_balance = 325.50;
   double college_debt = 5000;
   double *ptr_to_debt = nullptr;

    ...

  *ptr_to_debt = 0; // crashes now!
}
```

Class Challenge

Write a function called swap that accepts two pointers to integers and swaps the two values pointed to by the pointers.

Prize: 3 prize tickets (and maybe some candy)

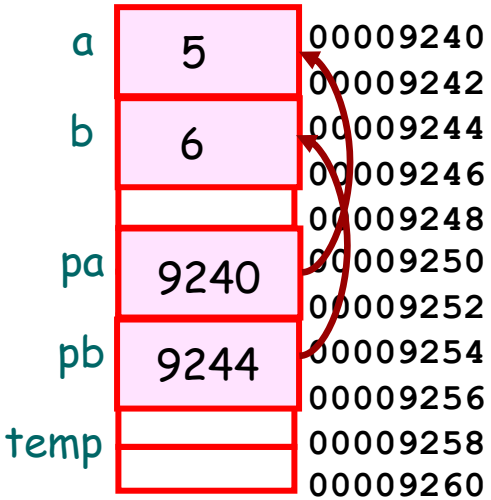
```
int main()
{
   int a=5, b=6;

   swap(&a, &b);
   cout << a; // prints 6;
   cout << b; // prints 5
}
```

Hint: Make sure that every time you define a pointer variable you immediately set its value to a valid address!!!

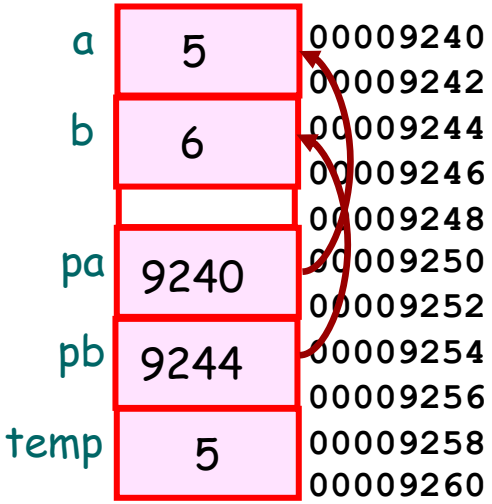
Class Challenge Solution

```
void swap (int *pa, int *pb)
{
   int temp;
   temp = *pa;    // #2
   *pa = *pb;     // #3
   *pb = temp;    // #4
}

int main()
{
    int a=5, b=6;
    swap(&a,&b); // #1
    cout << a;   // prints 6
    cout << b;   // prints 5
}
```

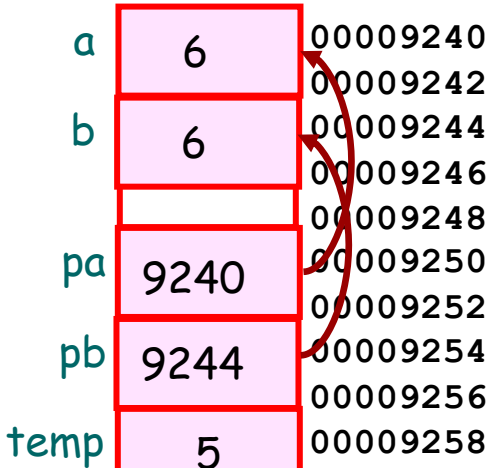
Line #1: We get the addresses of a (9240) and b (9244) and pass them to variables pa and pb in the swap() class.



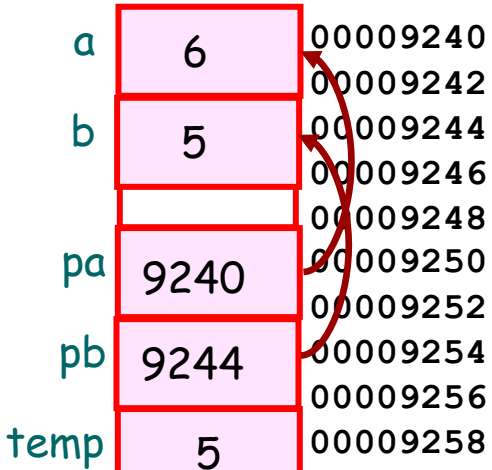
Line #2: We get the address stored in the pa variable, which is 9240. We then go to 9240 and get the value stored there, which is 5, and stick it into temp.



Line #3: We get the address stored in the pb variable, which is 9244. We then go to 9244 and get the value stored there, which is 6, and stick it into the location pointed to by pa, which is location 9240. This overwrites the a variable with 6.



Line #4: We get value in the temp variable, which is 5. We then stick 5 into the location pointed to by pb, which is location 9244. This overwrites the b variable with 5.

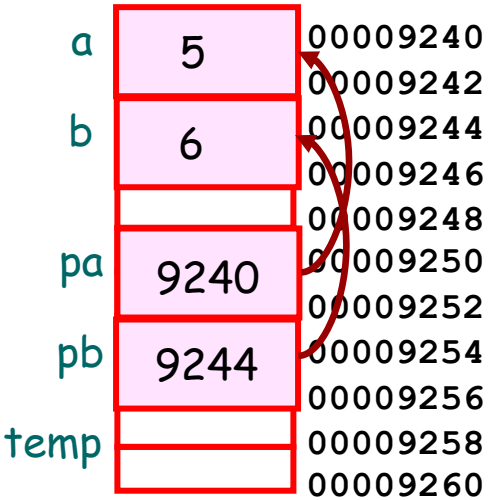


Wrong Solution #1

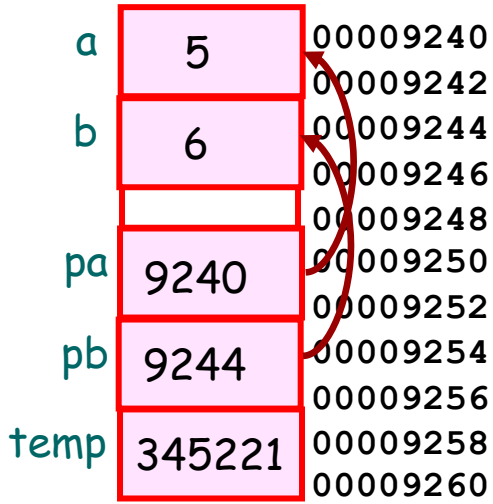
```
void swap (int *pa, int *pb)
{
  int *temp;      // #2
  *temp = *pa;    // #3
  *pa = *pb;
  *pb = *temp;
}

int main()
{
   int a=5, b=6;
   swap(&a,&b); // #1
   cout << a;
   cout << b;
}
```

- Notice that in this example, we defined the temp variable used in the swap() as a **pointer**:
  int *temp;
- Also notice that we never initialize the value of that temp pointer! So it will hold a random address and therefore point randomly in memory.
- That's bad!

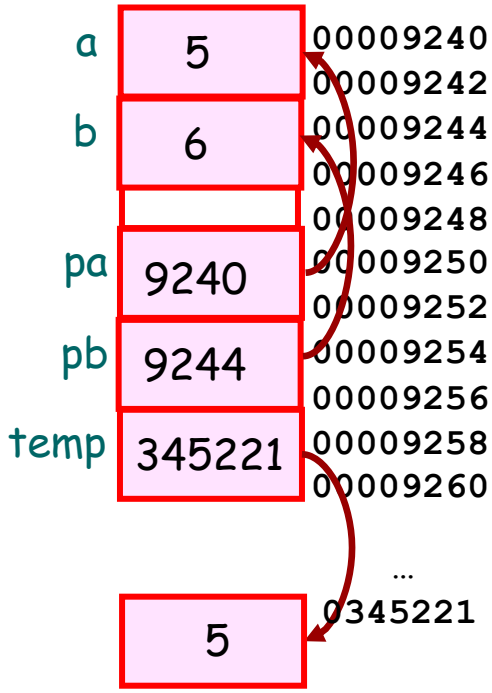
Line #1: We get the addresses of a (9240) and b (9244) and pass them to variables pa and pb in the swap() class.



Line #2: We define the temp pointer variable but FORGET to initialize its value so it will be random. In this case, the random address it holds is 345221.



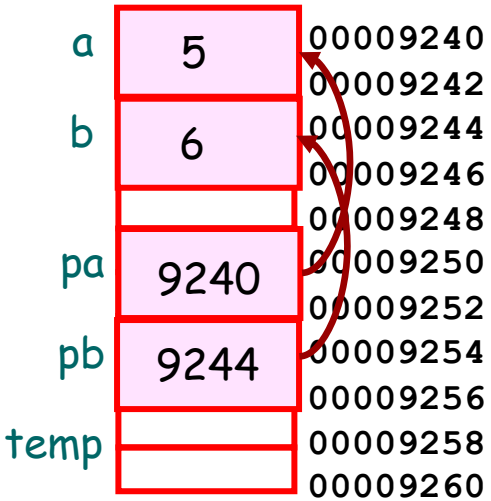
Line #3: We get the value pointed to by pa, which is 5. We then go to where the temp pointer points so we can store that value of 5. But wait! temp points at some random spot in memory (address 345221)! So we end up storing a value of 5 over some other part of our program at location 345221! If we're lucky the code will crash, but more likely, it'll just corrupt that memory and continue to run!



Wrong Solution #2

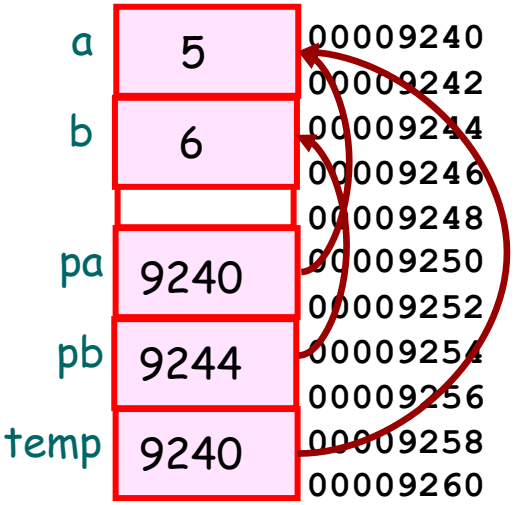
```
void swap (int *pa, int *pb)
{
  int *temp;
  temp = pa;    // #2
  pa = pb;      // #3
  pb = temp;    // #4
}

int main()
{
   int a=5, b=6;
   swap(&a,&b); // #1
   cout << a; // #5 prints 5
   cout << b; // prints 6
}
```

Line #1: We get the addresses of a (9240) and b (9244) and pass them to variables pa and pb in the swap() class.

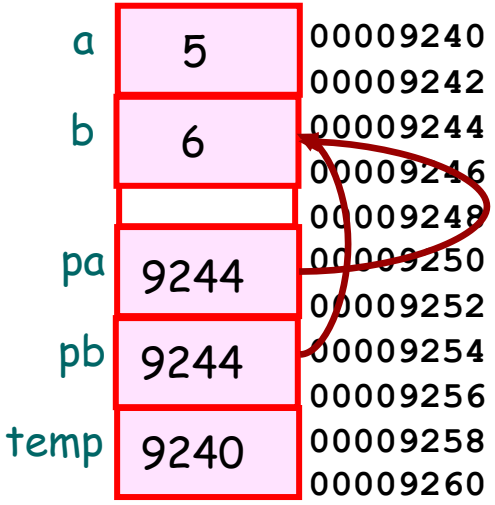


In this example, we never actually access what any of our pointers point to. We just swap the addresses held in the pointers themselves.

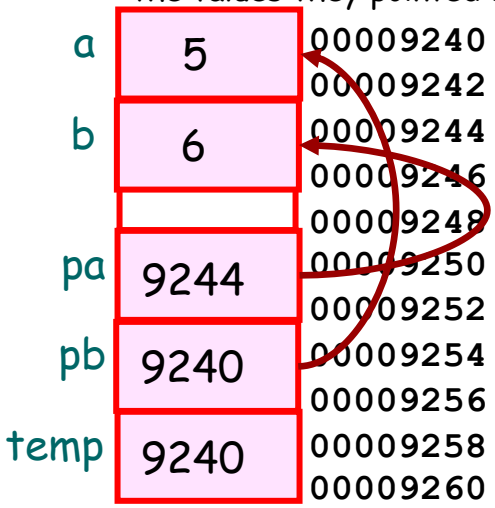
Line #2: We set the temp variable's value equal to the pa variable's value. So this puts the address 9240 into the temp pointer variable. temp also points to variable a now.



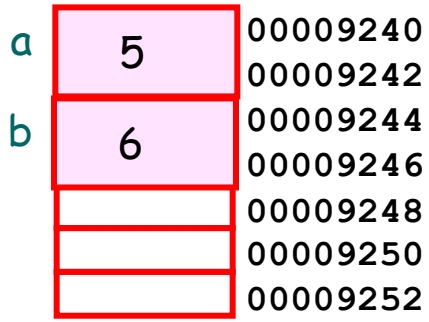
Line #3: We set pa's value to the value held in bp. So this copies 9244 from pb into pa. Now both pa and pb hold a value of 9244, and both pointers point to variable b.



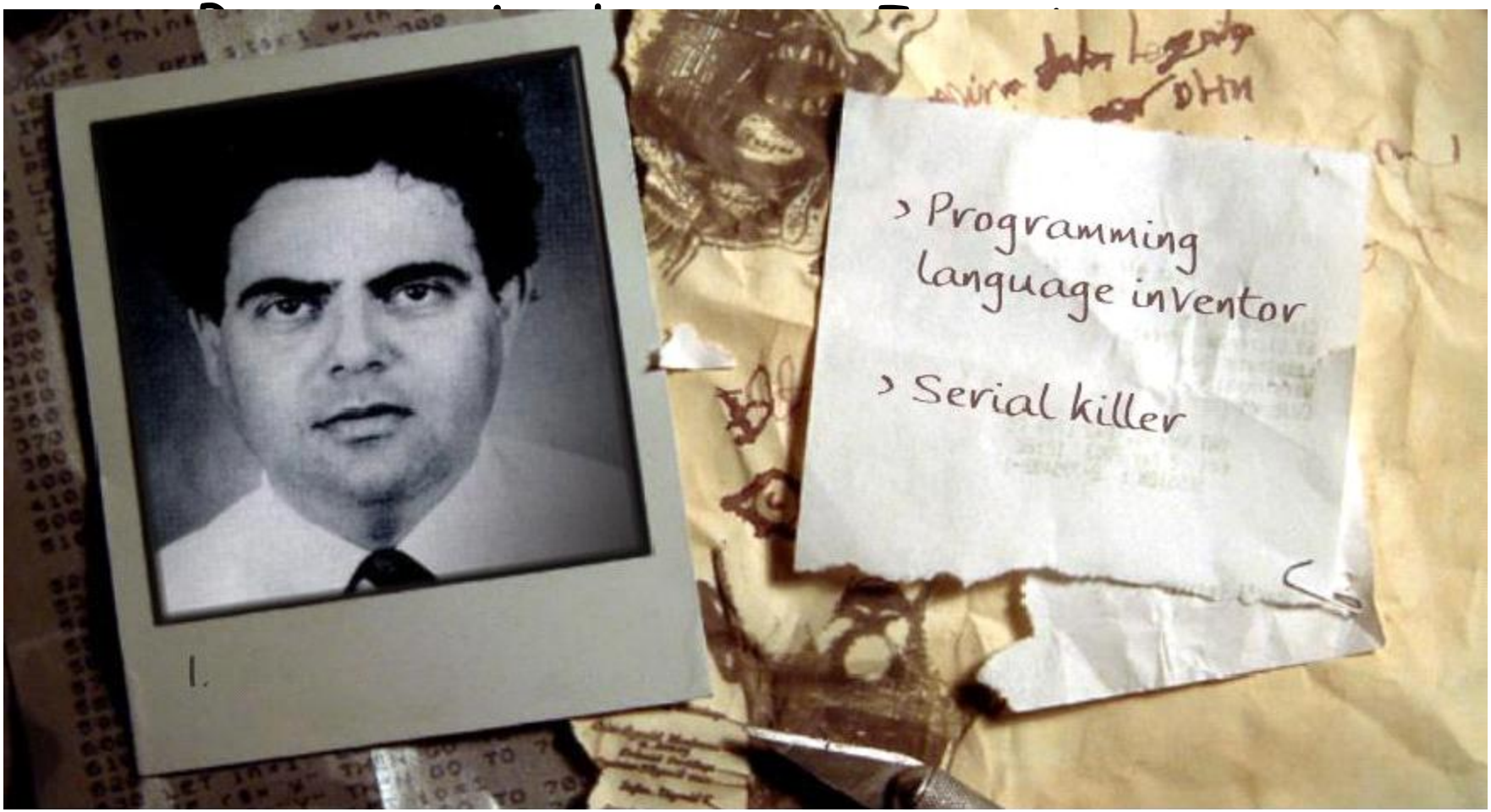
Line #4: We set pb's value to the value held in temp (9240). So now pb points at variable a. pa and pb have had their values swapped, but we never changed the values they pointed at!!!



Line #5: When we return to main(), our a and b variables have not changed at all!



Let's Play….

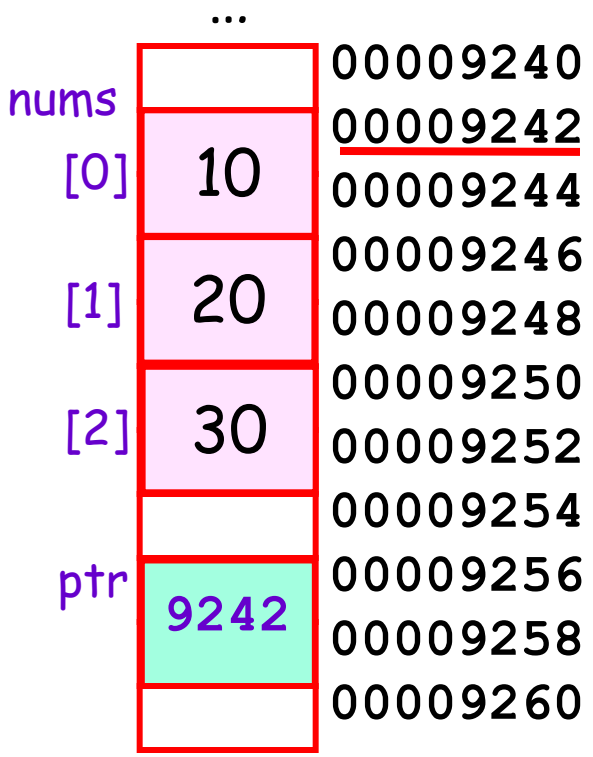


Arrays, Addresses and Pointers

- Just like any other variable, every array has an address in memory.
- So in our code below, nums is just an array. It holds three regular integer values. But it doesn't hold an address like a pointer variable, so it's not a pointer variable!
- While every array has an address, in C++ you don't use the & operator to get an array's address!
- You simply write the array's name (without brackets) and C++ will give you the array's address! See line #1.
- In contrast, the ptr variable is a valid C++ pointer variable.
- Line #2 defines the pointer variable ptr, and then sets its value to the address of the start of the nums array, which is 9242. So after line #2, ptr points to the start of the nums array.

```
int main()
{
   int nums[3] = {10,20,30};

   cout <<   nums;     // #1 prints 9242

   int *ptr = nums;   // #2


}
```

- Question: So is "nums" an address or a pointer or what?
- Answer: "nums" is just an array. But C++ lets you get its address without using the & so it looks like a pointer…



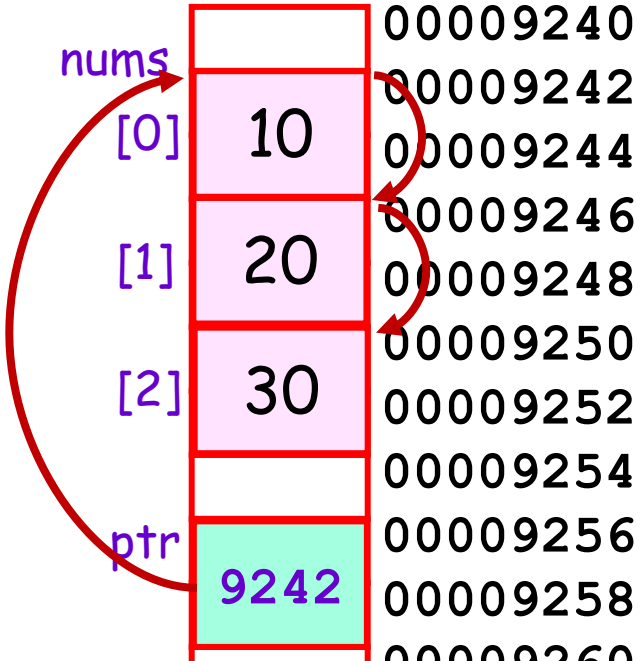
Arrays, Addresses and Pointers

- In C++, a pointer to an array can be used just as if it were an array itself!
- Or you can use the * operator with your pointer to access the array's contents.
- In C++, the two syntaxes have identical behavior:

ptr[j] ↔ *(ptr + j)

- They both mean: "Get the value in ptr, and go to that address in memory, then skip down j elements and get the value." When we say "skip down j elements," we don't mean skip j bytes, but j of whatever type of variable is stored in the array. So in the nums array, we'd skip down j ints.
- Line #1: Since ptr points to the top of the nums array, this prints out the value that is two integer elements from the top of the nums array (in the 3rd slot of the array). It works just as if ptr were a normal array with brackets!
- Line #2: Since ptr points to the top of the nums array, this prints out the value that is at address 9242 – i.e., the first element of the nums array. It's the same as cout << num[0] or cout << ptr[0].
- Line #3 says "print out the item that is two elements down from where ptr points." This is the same as printing out the value of ptr[2] or nums[2].

```
int main()
{
   int nums[3] = {10,20,30};

   int *ptr = nums;  // pointer to array

   cout << ptr[2];   // #1 prints nums[2] or 30
   cout << *ptr;     // #2 prints nums[0] or 10
   cout << *(ptr+2);// #3 prints nums[2] or 30

}
```



Pointer Arithmetic and Arrays

array 3000

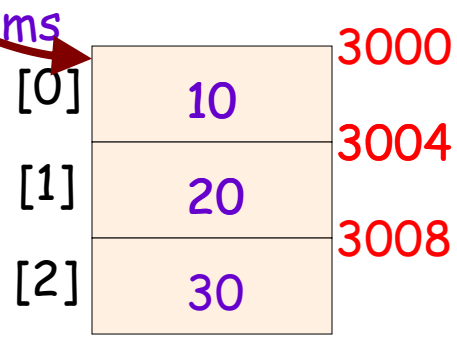
```
void printData(int array[ ])
{
    cout << array[0] << "\n"; // #2
    cout << array[1] << "\n"; // #3
}
```

```
int main()
{
   int nums[3] = {10,20,30};

   printData(nums);  // #1

   printData(&nums[1]);

   printData(nums+1);
}
```

- Did you know that when you pass an array to a function… you're really just passing the address to the start of the array … not the whole array itself!
- So on line #1, we take the address of the nums array (3000) and pass it to the printData() function. It's as if we wrote printData(&nums).
- The printData() function has a parameter that looks like an int array: int array[]
- But in reality, that "array" parameter is a pointer variable! You could have also defined that parameter like this (it'd work the same way):

  void printData(int *array)

- Why? C++ secretly translates int array[] to int *array when it compiles your code!
- So our prrintData() function has an array pointer parameter that gets the start address of the nums array (3000) when you run line #1. Notice its value is 3000 (upper-left)
- When you run line #2, it prints out the value at (3000 + 0) using the array bracket notation, which is the same as nums[0] or 10.
- When you run line #3, it prints out the value at (3000 + 1 integer down) using the array bracket notation, which is the same as nums[1] or 20.



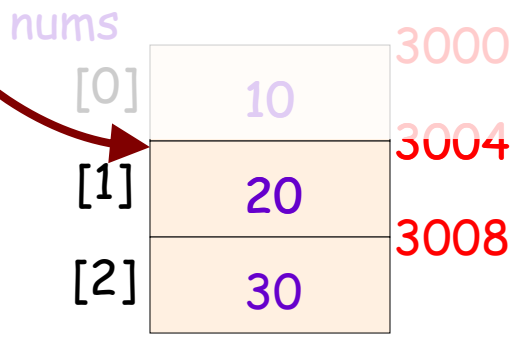
Pointer Arithmetic and Arrays

array 3004

```
void printData(int array[ ])
{
    cout << array[0] << "\n"; // #2
    cout << array[1] << "\n"; // #3
}
```

```
int main()
{
   int nums[3] = {10,20,30};

   printData(nums);

   printData(&nums[1]); // #1

   printData(nums+1);
}
```

- Now when we run line #1, this will pass the address of the second element of the nums array to our printData() function.
- Here it's a bit more obvious that you're just passing an address since we use the & syntax to get the address of nums[1]: &nums[1]
- This sends a value of 3004, the address of the second element of the array, to the printData() function and places this in the array parameter.
- From the perspective of the printData() function, it is processing an array that starts at 3004, not 3000, since all it has is a pointer to location 3004. It does not technically know the nums array starts at 3000!
- When it prints out array[0] on line #2, this prints out the value at 3004 (3004 + 0 elements down), which is nums[1], or 20.
- And when it prints out array[1] on line #2, this prints out the value at 3008 (3004 + 1 element down), which is nums[2], or 30. Isn't that wild? array[1] refers to nums[2]!
- So essentially the printData function is looking at a sub-segment of the array without knowing it!



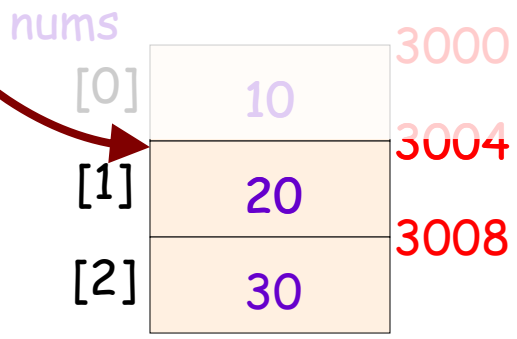
Pointer Arithmetic and Arrays

array 3004

```
void printData(int array[ ])
{
    cout << array[0] << "\n"; // #2
    cout << array[1] << "\n"; // #3
}
```

```
int main()
{
   int nums[3] = {10,20,30};

   printData(nums);

   printData(&nums[1]);

   printData(nums+1); // #1
}
```

- When we run line #1, this will also pass the address of the second element of the nums array to our printData() function.
- It's identical to the line above it that passes in &nums[1].
- Why? Well when we specify the name of an array without brackets, as we learned, C++ replaces this with the start address of the array (3000)
- Then we add 1 to it, taking this to 3004: (3000 + 1 integer element down).
- This sends a value of 3004, the address of the second element of the nums array, to the printData() function and places this in the array parameter.
- As before, line #2 will print out 20, and line #3 will print out 30.
- In case your wondering, if we changed line #2 to:
  cout << array[-1] << "\n";
- the call from line #1 would print out a value of 10, which is exactly 1 integer element up from location 3004.
- Wild, huh? Since the "array" variable is just a pointer, C++ doesn't care whether you add or subtract from it. It just computes the target address and accesses the value for you!

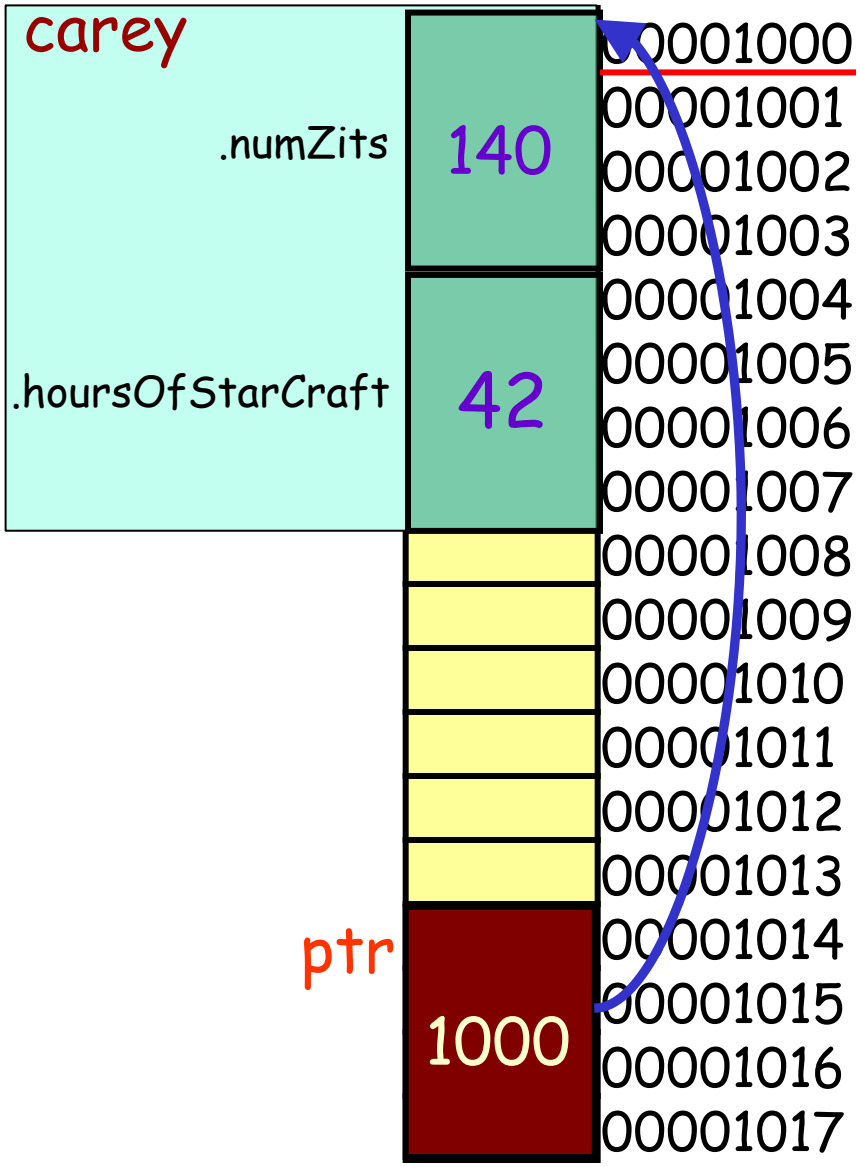


Pointers Work with Structures Too!

You can use pointers to access structs too! Use the * to get to the structure, and the dot to access its fields.

Or you can use C++'s -> operator to access fields!

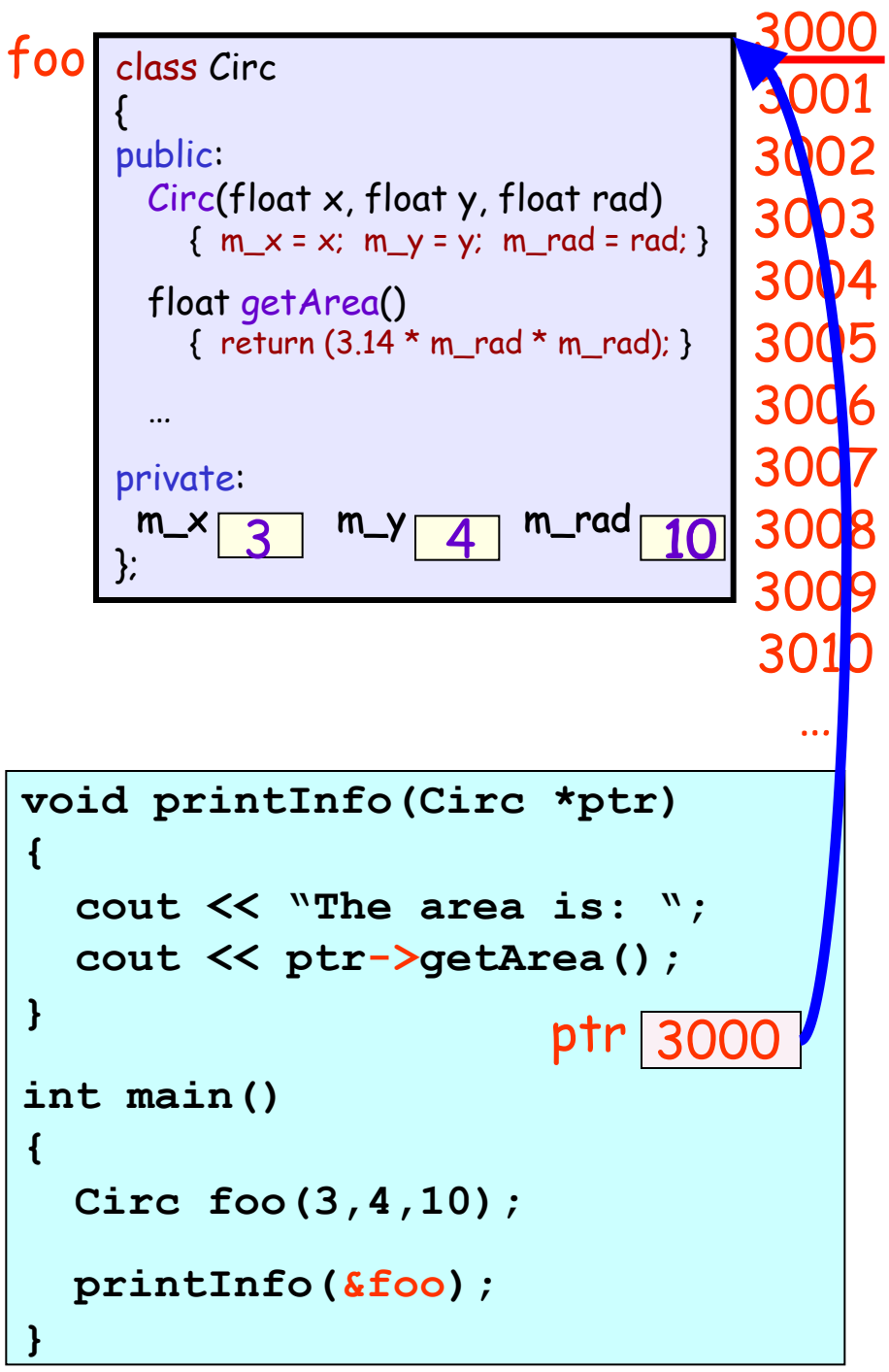
```
struct Nerd
{
  int numZits;
  int hoursOfStarCraft;
};

int main()
{
   Nerd  carey;
   Nerd  *ptr;

   ptr = &carey;

   (*ptr).numZits = 140;
    ptr->hoursOfStarCraft = 42;
}
```



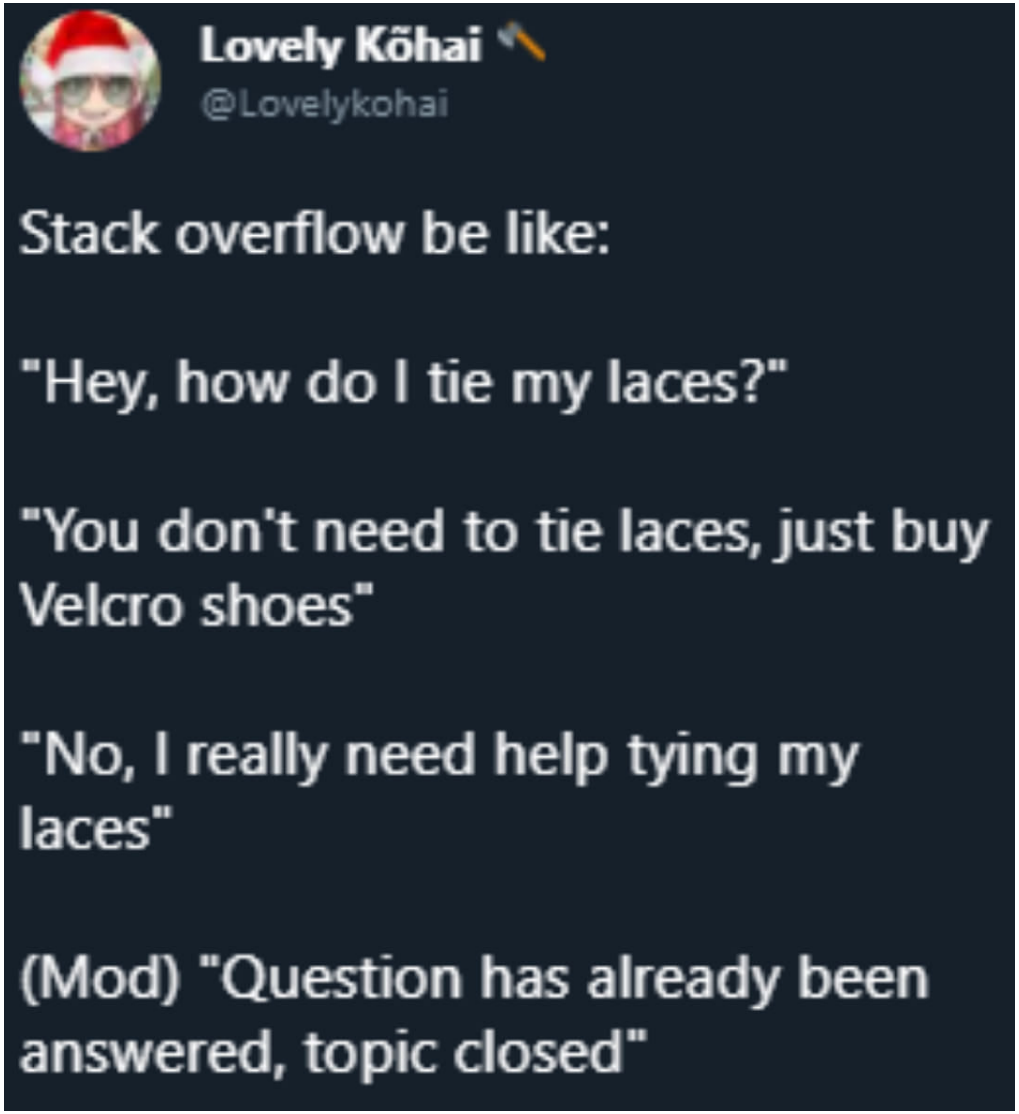
Classes and Pointers

```
class Circ
{
public:
   Circ(float x, float y, float rad)
      {  m_x = x;  m_y = y;  m_rad = rad; }

   float getArea()
      {  return (3.14 * m_rad * m_rad); }

    ...

private:
  float m_x, m_y, m_rad;
};
```

- You can use pointers with classes just like you do with structs.
- In this example, we define the foo variable which happens to be at address 3000.
- When we call printInfo(), we pass the address of foo, or 3000, which is copied into the ptr parameter.
- Then we can use ptr to call the member functions of our foo variable: ptr->getArea();
- We could also call foo's member functions using the * syntax like so: (*ptr).getArea();



Meme, anyone?


Lovely Kōhai
@Lovelykohai
Stack overflow be like:
"Hey, how do I tie my laces?"
"You don't need to tie laces, just buy Velcro shoes"
"No, I really need help tying my laces"
(Mod) "Question has already been answered, topic closed"

Classes and the "this" Pointer

Before C++, in the dark ages when Carey learned programming, we didn't use classes!

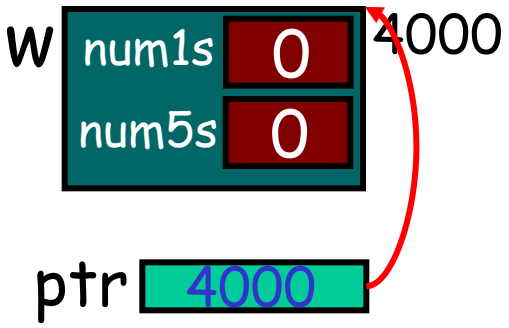
Let's see how we used to do things… with structs, pointers, and functions instead of classes!

And maybe this will help us understand how C++ classes actually work!

The Old Days…Before Classes

```
struct Wallet
{
  int num1s, num5s;
};

void Init(Wallet *ptr)
{
  ptr->num1s = 0;
  ptr->num5s = 0;
}

void AddBill(Wallet *ptr, int amt)
{
  if (amt == 1) ptr->num1s++;
  else if (amt == 5) ptr->num5s++;
}
```

```
void main()
{
  Wallet w;

  Init(&w);

  AddBill(&w , 5);
}
```



- This is how we used to do object-oriented programming before classes existed (in the C language, before C++)
- We'd define a struct with all of our relevant data members
- And then define a set of functions to operate on that struct.
- We'd always pass a pointer to the struct to each the functions, since references didn't exist in the C language back then.
- As it turns out, C++ classes work in an almost identical fashion! We'll see how classes work under the hood in a few slides.
- In this example, we have a Wallet struct and associated functions.
- An Init() method to initialize a wallet struct
- An addBill() method to add a bill to our wallet
- Note that both functions take a pointer to a Wallet variable as their parameter.

The Wallet Class

```
class Wallet
{
public:
    void Init();
    void AddBill(int amt);
    …
private:
    int num1s, num5s;
};

void Wallet::Init()
{
     num1s =    num5s = 0;
}
void Wallet::AddBill(int amt)
{
    if (amt == 1)      num1s++;
    else if (amt == 5)num5s++;
}
```

Here's a class equivalent of our old-skool Wallet…

As you can see, we can initialize a new wallet…

And we can add either a $1 or $5 bill to our wallet.

Our wallet then keeps track of how many bills of each type it holds…

```
int main()
{
  Wallet a;

  a.Init();
  a.AddBill(5);
}
```

Classes and the "this" Pointer

- As it turns out, C++ basically converts your class into the same type of C code that we saw earlier with a struct and functions that take a pointer to the struct as an argument.
- On the left, we see the C++ class we defined for a Wallet
- On the right, we see the code that C++ actually translates your code into!
  Notice how C++ adds a hidden first argument that's a pointer to your original variable!
  That hidden first argument is always called "this" – yes, the variable is called "this"!

Here what your Init() method looks like...

```
void Wallet::Init()
{
    num1s =   num5s = 0;
}
void Wallet::AddBill(int amt)
{
   if (amt == 1)      num1s++;
   else if (amt == 5)num5s++;
}
...
```

But here's what's REALLY happening! ☺

```
void Init(Wallet *this)
{
   this->num1s =  this->num5s = 0;
}
void AddBill(Wallet *this,int amt)
{
  if (amt == 1)       this->num1s++;
  else if (amt == 5) this->num5s++;
}
...
```

So here we're calling the Init() method of a...

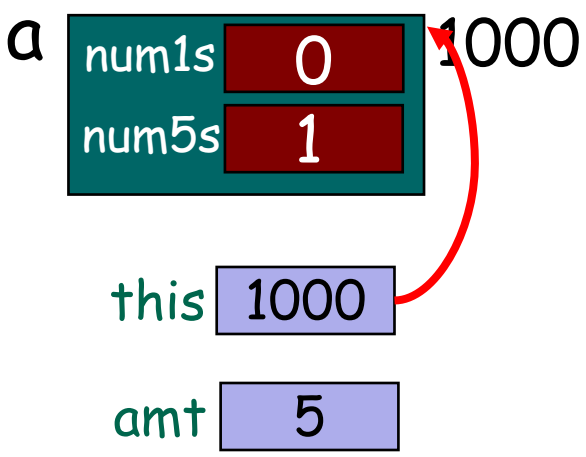
```
int main()
{
  Wallet a, b;

  a.Init();
  b.AddBill(5);
}
```

But here's what's REALLY happening! ☺

```
int main()
{
  Wallet a, b;

  Init( &a );
  AddBill(&b, 5);
}
```

Classes and the "this" Pointer

```
void Wallet::Init(Wallet *this)
{
  this->num1s = this->num5s = 0;
}
void Wallet::AddBill(Wallet *this, int amt)
{
   if (amt == 1)     this-> num1s++;
   else if (amt==5)  this-> num5s++;
}
...
```




```
int main()
{
  Wallet a;
          1000
  a.Init(&a);
               1000  5
  a.AddBill(&a , 5);
}
```

This is how it actually works under the hood….

But C++ hides the "this pointer" from you to simplify things.

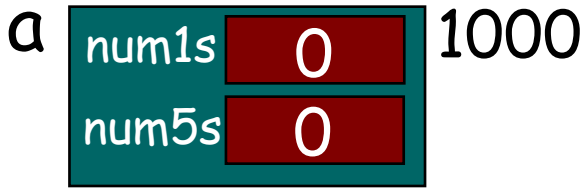
Classes and the "this" Pointer

You can explicitly use the "this" variable in your methods if you like! It works fine!

```
void Wallet::Init()
{
    this->num1s = this->num5s = 0;
    cout << "I am at address: " << this;
}
void Wallet::AddBill(int amt)
{
    if (amt == 1)              num1s++;
    else if (amt == 5)         num5s++;
}
...
```

```
int main()
{
  Wallet a;

  a.Init();
  cout << "a is at address: " << &a;
}
```

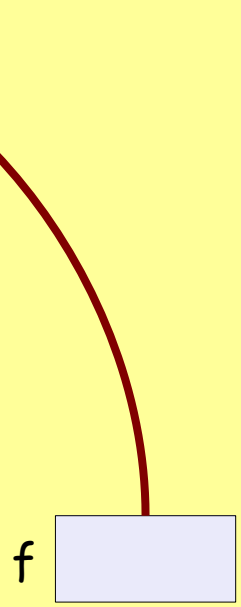
- While C++ hides the "this pointer" from you, if you want, your class's methods can explicitly use it.
- Your class's methods can use the this variable to determine their address in memory!
- For instance, you can write **this->num1s = 0;** and it will work the same as **num1s = 0;**
- So now you know how C++ classes work under the hood!
- Notice that in this example, our Wallet variable a is at address 1000 in memory.
- When we run our Init() method, it prints out the a variable's address (1000)
- And we also print out a's address in main, and see it's also 1000.



I am at address: 1000
a is at address: 1000

Pointers… to Functions?!?

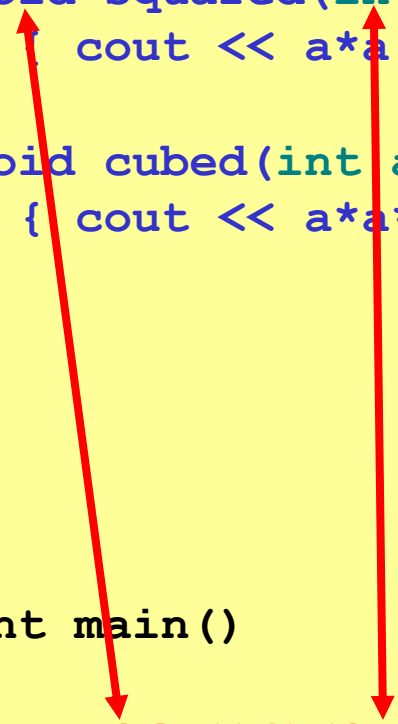
```
3000 void squared(int a)
3050   { cout << a*a;}
3100
3150 void cubed(int a)
3200   { cout << a*a*a;}
3250
3300
3350
3400
3450
3500
3550 int main()
3600 {
3650    FuncPtr f;
3700
3750    f = &squared;   // #1
3800    f(10);          // #2
3850    f = &cubed;     // #3
3900    f(2);           // #4
3950 }
```



- Just like every variable, every function has an address in memory too!
- YES! Just as you can have pointers to variables, in C++ you can also have pointers to functions!
- Let's gloss over the syntax for a second, and just see how it might work…
- First we define a function pointer like variable f.
- The function pointer variable can hold the address of… a function!
- On line #1, we set the value of variable f to the address of the squared() function.
- So f now points at the squared() function.
- Once we do this, we can use variable f just like it's a regular function! See line #2 how we call f(10); This calls the squared function and passes in 10!
- And like any variable, we can change a function pointer's value. On line #3 we set the value of pointer f to the address of the cubed function.
- f no longer points at squared(), but not points to cubed()
- So when we call f(2) on line #4, this will pass a value of 2 to the cubed() function and print out 8.
- Ok, let's see the real syntax.

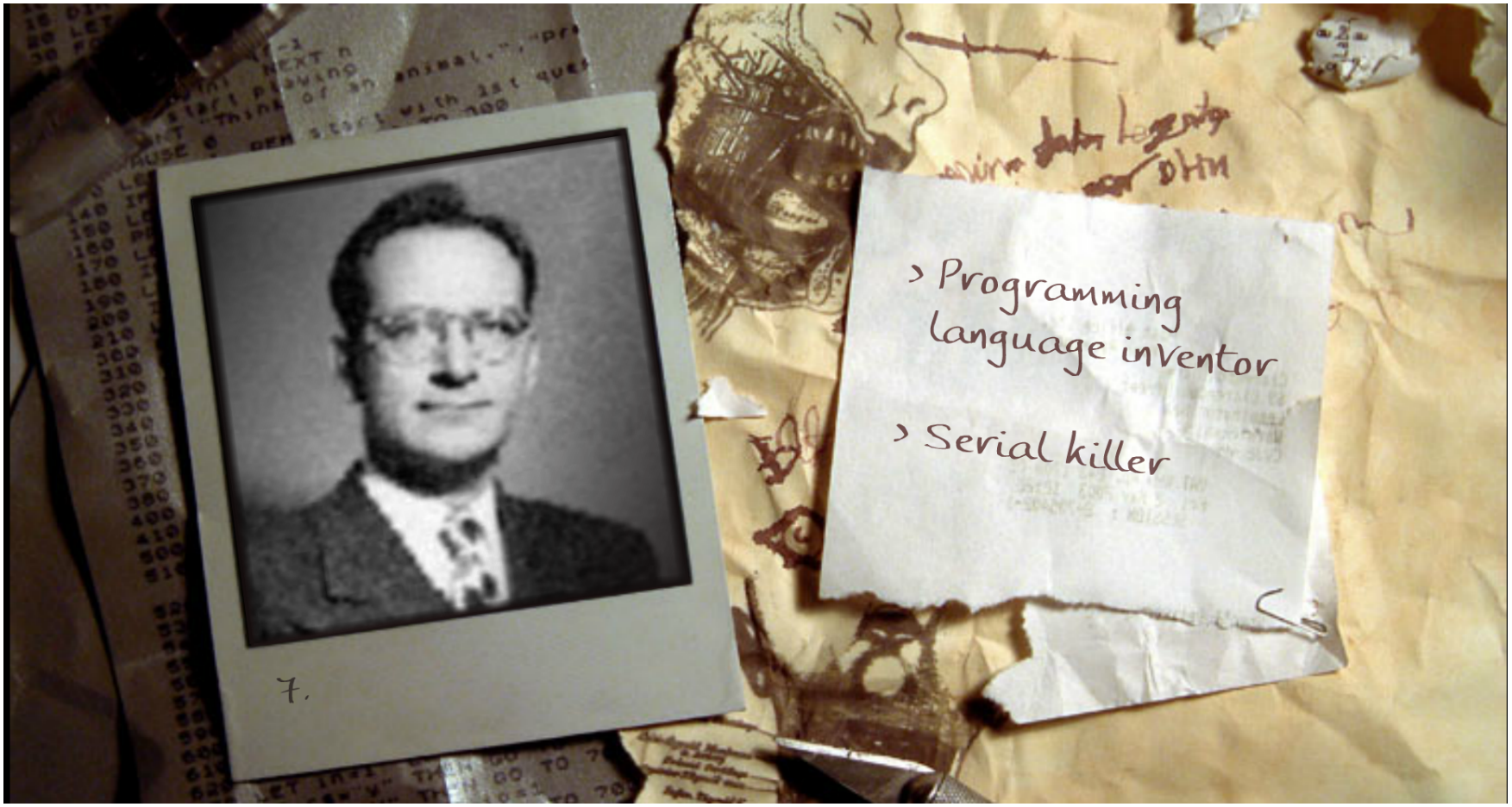
Pointers… to Functions?!?

```
3000 void squared(int a)
3050   { cout << a*a;}
3100
3150 void cubed(int a)
3200   { cout << a*a*a;}
3250
3300
3350
3400
3450
3500
3550 int main()
3600 {
3650     void (*f)(int);   // #1
3700
3750     f = &squared;
3800     f(10);
3850     f = &cubed;
3900     f(2);
3950 }
```



- Ok, here's the real syntax to define a function pointer called f on line #1.
- You start by specifying the return type of the functions that you want your pointer f to be able to point to.
- In this case, both squared() and cubed() return nothing – they have a void return type.
- So we use "void" as our return type on line #1
- Next you write the pointer variable name, in this case, f, in parenthesis with an asterisk * in front of it, e.g.:
  (*f) or (*func_ptr) or (*bletch)
- Next you specify the types of the parameters of the functions that you want f to be able to point to.
- In our example, squared() and cubed() both require an int parameter, so we define our function pointer f with a single int parameter: (int)
- If your function took an int and a double, then you'd write (int, double)
- So in total, we define f as we see on line #1: void (*f)(int);
- Variable f can only be set to point to functions that have a void return type, and that take one int parameter!
- Oh, and you don't need to use an ampersand & in front of your function name to get its address. Like getting the address of an array, you can omit the & if you like.

And now it's time for your favorite game!



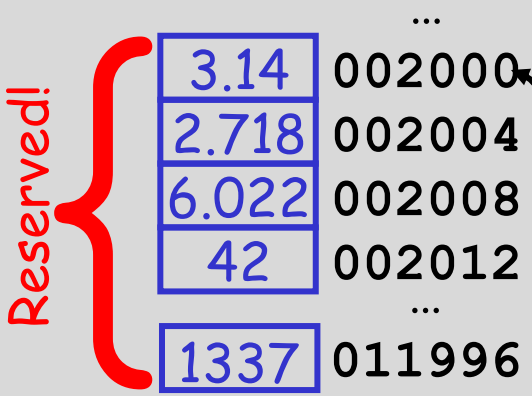
Dynamic Memory Allocation…
What's the big picture?

Often a program won't know how much memory it needs to solve a problem until it's actually running.

In these cases, C++ lets you reserve a chunk of memory on-demand, and gives you a pointer to it.

Your code can use the memory via the pointer.

When you're done, you then ask C++ to unreserve it!




```
void computeSomethingImportant(int count)
{
    ptr = askC++ForThisMuchMemory(count);
    operateOnTheMemoryAt(ptr);
    tellC++WereDoneWithThisMemory(ptr);
}
```

ptr
2000

Uses:

Dynamic allocation is used in video games, word processors, search engines, etc., etc.

New and Delete (For Arrays)

Let's say we want to define an array, but we won't know how big to make it until our program actually runs …

```
int main()
{
    int *arr;
    int size;

    cin >> size;

    arr = new int[size];

    arr[0] = 10;
    arr[2] = 75;

    delete [] arr;
}
```

#1 #2 #3 #4 #5

Note: Don't forget to include brackets: delete [ ] ptr;

if you're deleting an array…

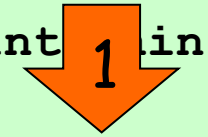
The new command can be used to allocate an arbitrary amount of memory for an array.

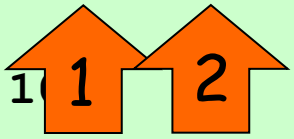
How do you use it?

1. First, define a new pointer variable.
2. Then determine the size of the array you need.
3. Then use the new command to reserve the memory. Your pointer gets the address of the memory.
4. Now use the pointer just like it's an array!
5. Free the memory when you're done.

New and Delete (For Arrays)

```
int main()
{
    int size, *arr;

    cout << "how big? ";
    cin >> size;

    arr = new int[size];

    arr[0] = 10;
    // etc

    delete [] arr;
}
```





The new command requires two pieces of information:

1. What type of array you want to allocate.
2. How many slots you want in your array.

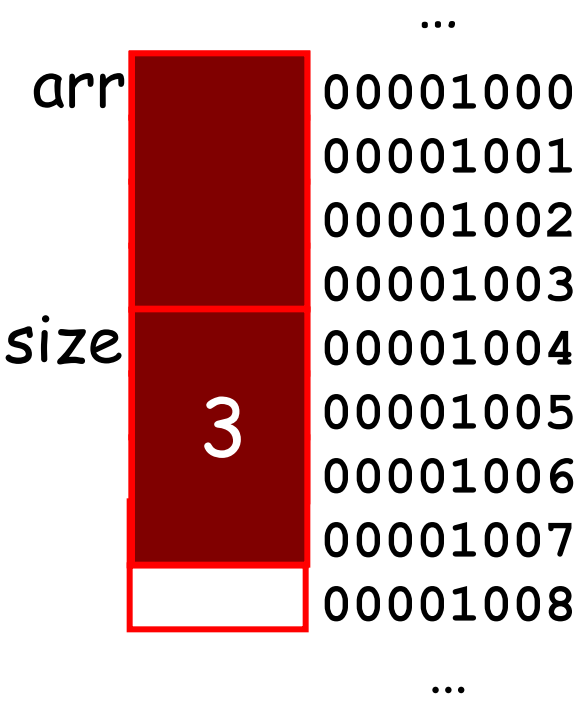
Make sure that the pointer's type is the same as the type of array you're creating!

New and Delete (For Arrays)

```
int main()
{
    int *arr;
    int size;

    cin >> size;

    arr = new int[size];

    arr[0] = 10;
    // etc

    delete [] arr;
}
```

Let's say the user types in a value of 3

4 * 3 = 12 bytes



- First, the new command determines how much memory it needs for the array.
- In the above example, the new command is trying to allocate an array of integers.
- In C++, a single integer (int) is 4 bytes long.
- So if the user specified a size value of 3, that means the array will require 4 * 3 bytes total

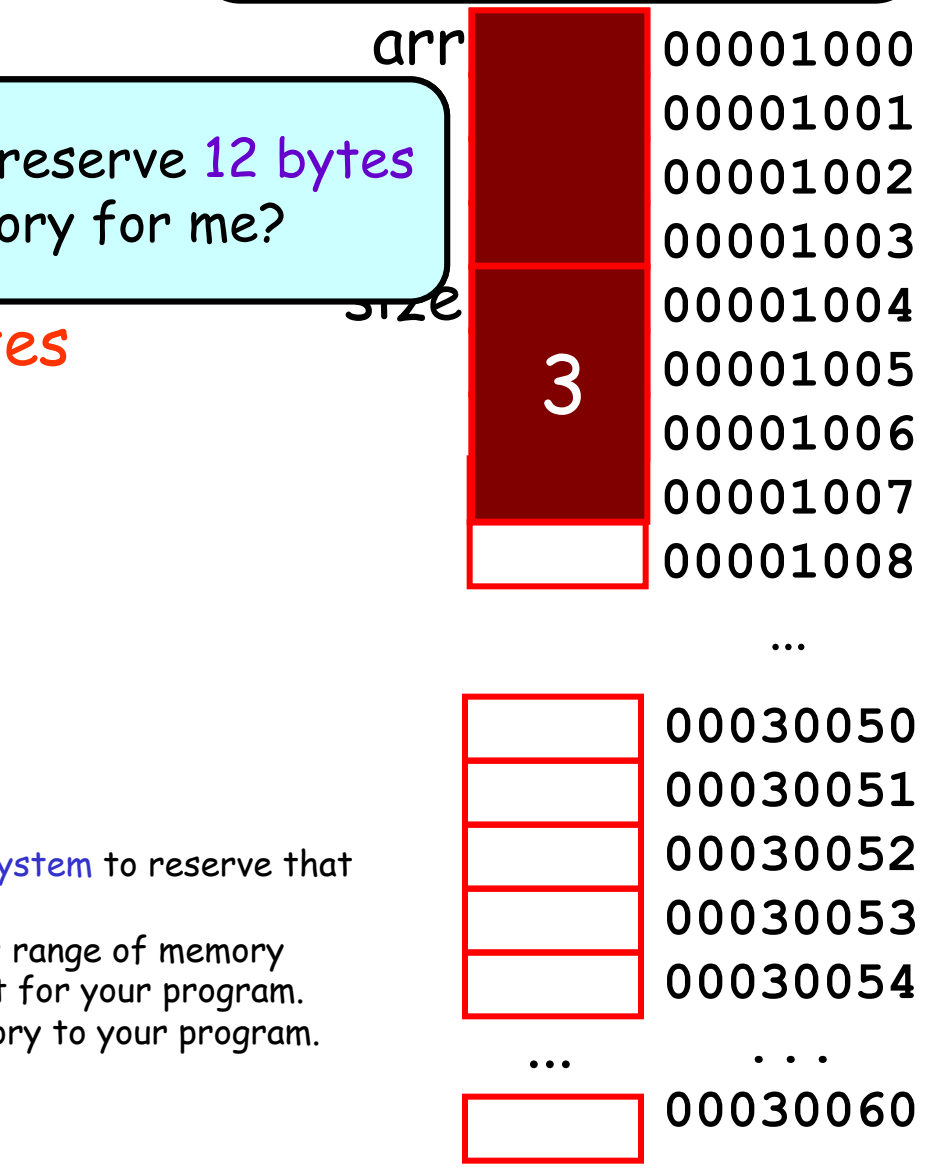
New and Delete (For Arrays)

```
int main()
{
    int *arr;
    int size;

    cin >> size;

    arr = new int[size];

    arr[0] = 10;
    // etc

    delete [] arr;
}
```

C++: can you reserve 12 bytes of memory for me?

4 * 3 = 12 bytes

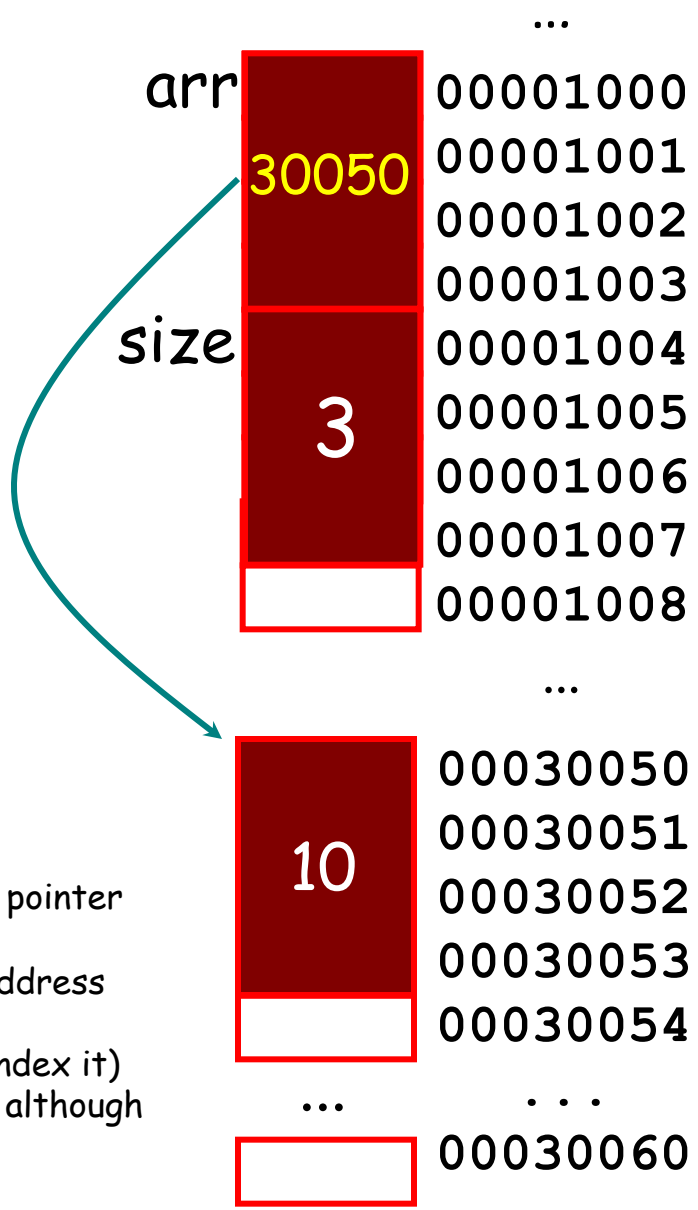
Operating System: I found 12 bytes of free memory at address 30050.



- Next, the new command asks the operating system to reserve that many bytes of memory.
- The operating system will locate a contiguous range of memory that's not currently being used and reserve it for your program.
- The OS will return the address of that memory to your program.

New and Delete (For Arrays)

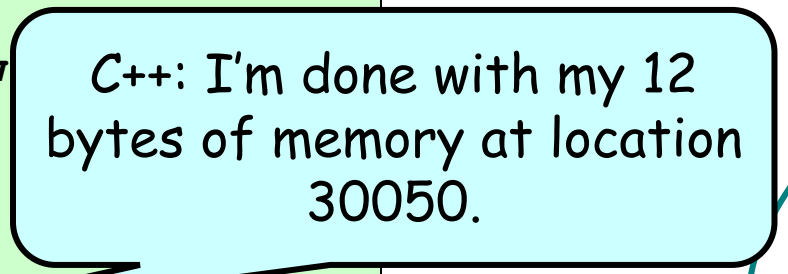
```
int main()
{
    int *arr;
    int size;

    cin >> size;

    arr = new int[size];

    arr[0] = 10;
    *(arr+1) = 20;  // arr[1] = 20;

    delete [] arr;
}
```

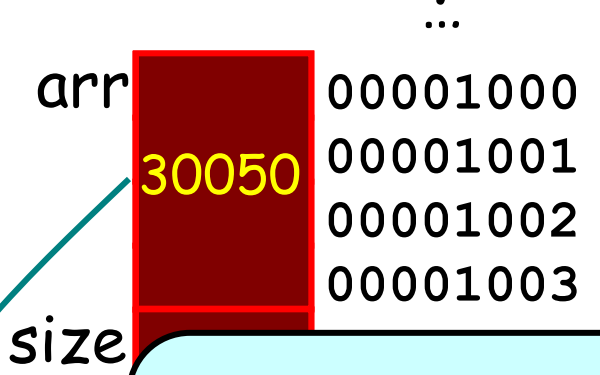


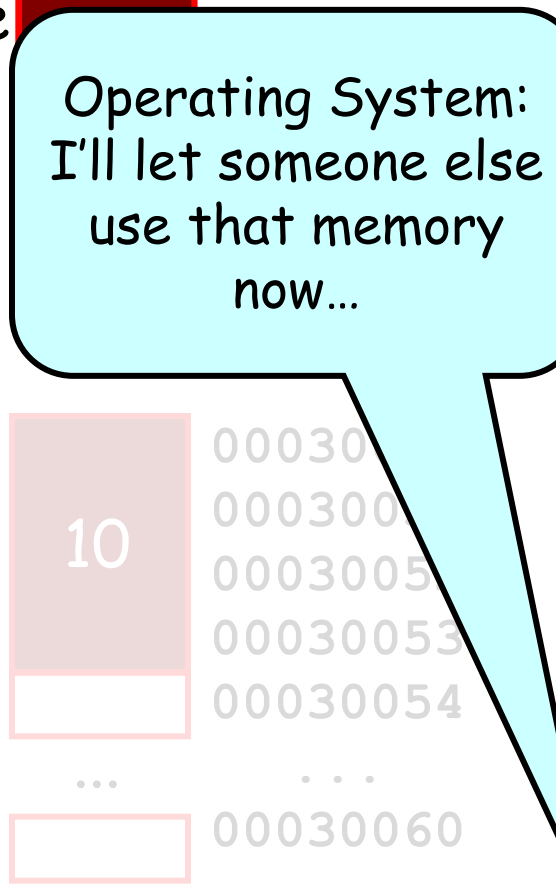
- Finally, C++ stores the address of the reserved memory in your pointer variable (arr, in this example)
- See how the arr variable holds a value of 30050, which is the address of the reserved memory.
- You can now treat your pointer just like an array (i.e. use [ ] to index it)
- You can also use the * notation if you like (instead of brackets), although brackets are preferred

New and Delete (For Arrays)

```
int main()
{
    int *arr;
    int size;

    cin >> size;

    arr = new

    arr[0] =
    // etc

    delete [] arr;
}
```







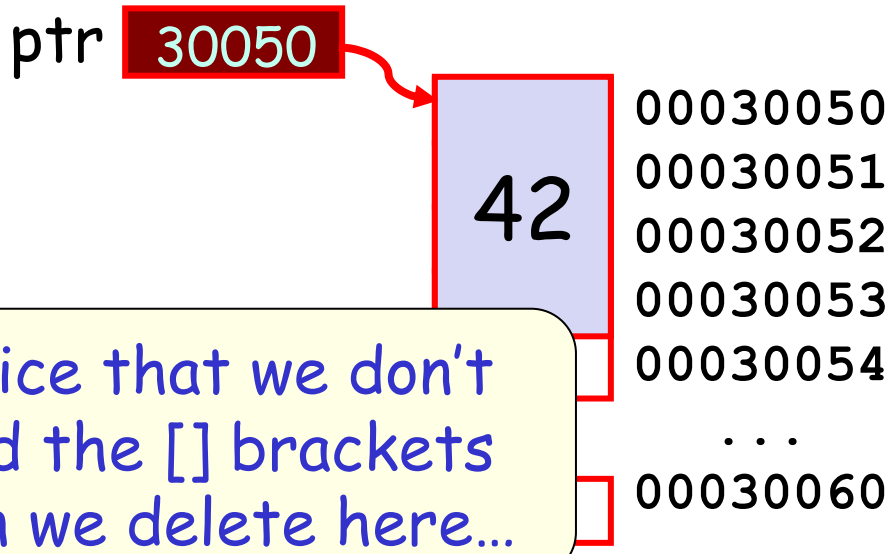
- When you're done using your memory, you use the delete command to free it. Don't forget the brackets [] if you're deleting an array!
- The delete command tells the OS to free the chunk of memory that was allocated so another program (or different part of your program) can use it.
- Note: When you use the delete command, you free the pointed-to memory (e.g., the block of 12 bytes at 30050), not the pointer variable itself! Our pointer variable (arr) still holds the address of the previously-reserved memory slots!
- But even though your pointer still contains an address, you are NOT allowed to use it after using the delete command, since it may now point to memory that's reserved by someone else!

New and Delete (For Non-Arrays)

- We can also use new and delete to dynamically create other types of variables as well!
- For instance, we can allocate an integer variable like this…

```
int main()
{
    // define our pointer
    int *ptr;

    // allocate our dynamic variable
    ptr = new int;

    // use our dynamic variable
    *ptr = 42;
    cout << *ptr << endl;

    // free our dynamic variable
    delete ptr;
}
```

Since we didn't allocate an array up here!



Notice that we don't need the [] brackets when we delete here…

New and Delete (

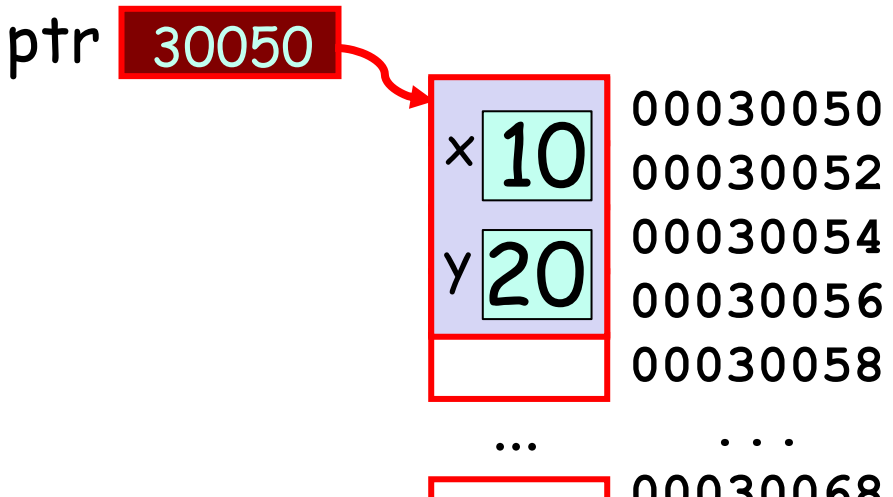
We can also use new and
create other types

```
struct Point
{
   int x;
   int y;
};
```

```
int main()
{
    // define our pointer
    Point *ptr;

    // allocate our dynamic variable
    ptr = new Point;

    // use our dynamic variable
    ptr->x = 10;
    (*ptr).y = 20;

    // free our dynamic variable
    delete ptr;
}
```

- Or we can allocate a struct variable like this…
- Notice how we can use the -> and * operators to access the structure being pointed to!

```
ptr->x = 10;
(*prt).y = 20;
```



New and Delete (I

We can also use new and
create other types

```
int main()
{

    // define our pointer
    Nerd *ptr;

    // allocate our dynamic variable
    ptr = new Nerd(150, 1000);

    // use our dynamic variable
    ptr->saySomethingNerdy();


    // free our dynamic variable
    delete ptr;
}
```

```
class Nerd
{
  public:
     Nerd(int IQ, int zits)
     {
        m_myIQ = IQ;
        m_myZits = zits;
     }
     void saySomethingNerdy()
     {
        cout << "C++ rocks!";
     }
   ...
};
```

- We can even allocate a class instance like this…
- In this example, the new command allocates enough memory to hold a single Nerd object.
- You could also allocate an array of Nerds if you wanted, so long as the class has a default c'tor:
  Nerd *arr_of_nerds = new Nerd[100];
- Once the OS reserves some memory and returns its address, C++ automagically calls the class's constructor to initialize the memory for you!
- Finally, once the object is fully constructed, the address of the object is returned to you and placed in your pointer variable (e.g., in ptr)
- When you delete the object, C++ first runs the destructor on the object for you, then asks the OS to free the memory.

```
// original class w/fixed array
class PiNerd
{
public:
   PiNerd() {
      for (int j=0; j< 10 ;j++)
       m_pi[j] = getPiDigit(j);
  }

   void showOff(){
     for (int j=0;j< 10  ;j++)
       cout << m_pi[j] << endl;
   }
private:
    int m_pi[10];
};
```

```
// new class with dynamic array
class PiNerd
{
pub PiNerd(int n) {
    PiNerd() {
      m_pi = new int[n]; // alloc array
      m_n = n;           // store its size!
      for (int j=0;j< 10  ;j++)
        m_pi[j] = getPiDigit(j);
  }
  ~PiNerd() {
    delete [] m_pi; //free memory
  }

   void showOff(){
     for (int j=0;j< m_n ;j++)
       cout << m_pi[j] << endl;
   }
private:
    int *m_pi, m_n;
};
```

Step #2: Update our c'tor so the user can pass in the size of the nerd's array.

Step #3: Use the new command to allocate an array of the right size. Remember its size in m_n!

Step #4: Update loop so we print all N #s.

Step #5: Add a destructor that frees the dynamic array when we're done!

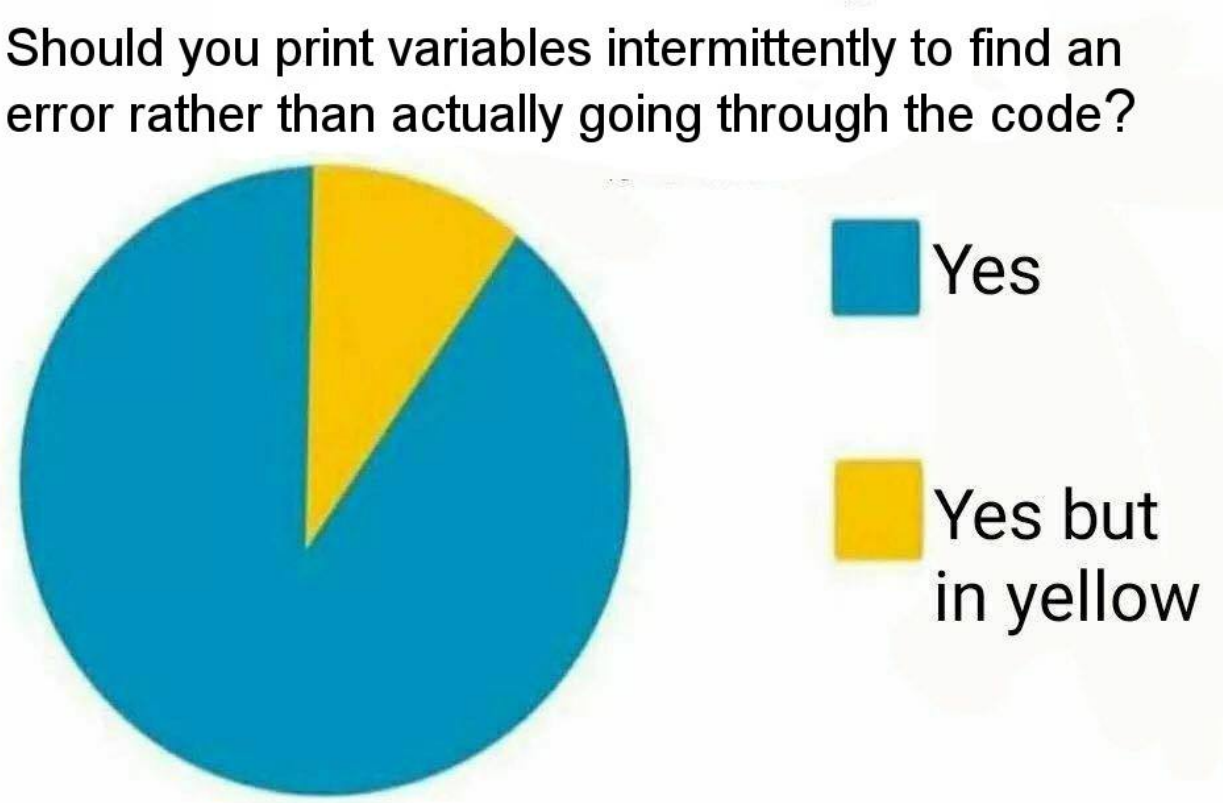
Step #1: Change our fixed array to a pointer variable and add a size variable.

```
int main() {
  PiNerd superNerdy(500);

  superNerdy.showOff();
}
```

Using new and delete in a class

(Update our PiNerd class from storing a fixed # of pi digits to any number of pi digits)

Copy Construction



* Note: This meme has nothing to do with copy construction.

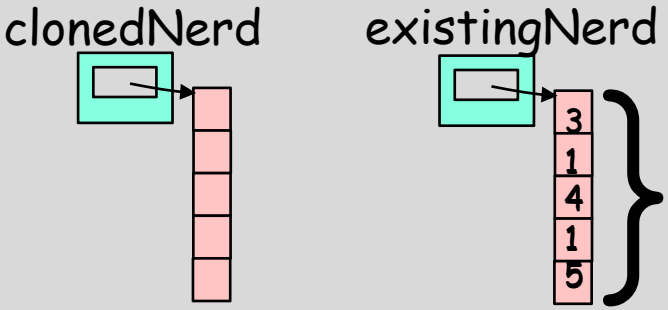
Copy Construction…
What's the big picture?

Copy construction is when we create (construct) a new object by copying the value of an existing object.

```
void cloneANerd()
{
   PiNerd existingNerd(4); // knows PI to 4 digits
   ...

   PiNerd clonedNerd = existingNerd;

   clonedNerd.showOff();  // prints 3.141
}
```

Create a new variable

By copying an existing variable



To clone more complex objects, we need to create a special function to do this, called a copy constructor.

Uses:

Copy constructors are required if you want to let users make a copy of more complex class variables.

Copy Construction

```
class Circ
{
public:
  Circ(float x, float y, float r)
  {
    m_x = x; m_y = y;  m_rad = r;
  }

  Circ(const Circ& old) // #1
  {
    m_x = old.m_x;   // #2
    m_y = old.m_y;   // #3
    m_rad = old.m_rad; // #4
  }

  float GetArea() const;
private:
  float m_x, m_y, m_rad;
};
```

- Just as we can define a constructor that initializes a new Circ variable based on (x,y) and radius values…
- We can also define a constructor that initializes a new Circ variable based on an existing Circ variable.
- How, we define a special constructor that takes in another Circle that we want to copy from as a parameter (#1)
- In its simplest form, this constructor just copies the member variable values from the original object to the new object (as shown in the constructor to the left - #2 - #4)
- In C++ lingo, this function is called a "copy constructor."

```
int main()
{
  Circ a(1, 2, 3);

  Circ b(a);
}
```

Creates a new circle…

By copying an existing circle!

Copy Construction Privacy Concerns?

b

```
class Circ
{
 ...
 Circ( const Circ &old )
 {
   m_x = old.m_x;
   m_y = old.m_y;
   m_rad = old.m_rad;
 }
 ...
private:
  m_x  m_y  m_rad
}
```

a

```
class Circ
{
 ...
 Circ(float x, float y, float rad)
 {
   m_x = x;
   m_y = y;
   m_rad = rad;
 }
 ...
private:
  m_x 1  m_y 2  m_rad 3
}
```

- But wait! Circ variable b is accessing the private variables/functions of Circ variable a – isn't that violating C++ privacy rules?
- That's not a problem. Every Circ variable is allowed to "touch" every other Circ variable's privates – "private" protects one class from another, *not* one variable from another (of the same class)!
- So every CSNerd object can touch every other CSNerd object's privates.
- But a CSNerd can't touch an EENerd's privates (for obvious reasons).

```
int main()
{
   Circ a(1,2,3);

   Circ b(a);
   ...
}
```

This means: "Initialize variable b based on the value of variable a."

Copy Construction

```
class Circ
{
public:
  Circ(float x, float y, float r)
  {
     m_x = x; m_y = y; m_rad = r;
  }
  Circ( const Circ & old)
  {
    m_x = oldVar.m_x;
    m_y = oldVar.m_y;
    m_rad = oldVar.m_rad;
  }
  float GetArea()
  {
    return(3.14159*m_rad*m_rad);
  }
private:
  float m_x, m_y, m_rad;
};
```

- The parameter to your copy constructor *should* be const! This is a promise that you won't modify the old variable while constructing your new variable - you're just reading its value!
- The parameter to your copy constructor *must* be a reference!
- The type of your parameter must be the same type as the class itself!

Copy Construction

```
class Circ
{
public:
  Circ(float x, float y, float r)
  {
    m_x = x; m_y = y; m_rad = r;
  }
       const Circ & old)
  {
    m_x = oldVar.m_x;
    m_y = oldVar.m_y;
    m_rad = oldVar.m_rad;
  }
  float GetArea()
  {
    return(3.14159*m_rad*m_rad);
  }
private:
  float m_x, m_y, m_rad;
};
```

Oh, C++ also allows you to use a simpler syntax…

Instead of writing:

Circ b(a);

which is ugly…

You can write:

Circ b = a;

It does exactly the same thing! It defines a new variable b and then calls the copy constructor!

```
int main()
{
   Circ a(1,2,3);

→Circ b = a; // same!
}
```

Copy Construction

```
class Circ
{
public:
  Circ(float x, float y, float r)
  {
     m_x = x; m_y = y; m_rad = r;
  }
  Circ(const Circ & old)
  {
    m_x = oldVar.m_x;
    m_y = oldVar.m_y;
    m_rad = oldVar.m_rad;
  }
  float GetArea()
  {
     return(3.14159*m_rad*m_rad);
  }
private:
  float m_x, m_y, m_rad;
};
```

The copy constructor is not just used when you initialize a new variable from an existing one:

Circ b(a);

It's used *any time* you make a new copy of an existing class variable.

Can anyone think of other times when a copy constructor would be used?

Copy Construction

temp

```
class Circ
{
 ...
 Circ(const Circ &old)
 {
   m_x = old.m_x;
   m_y = old.m_y;
   m_rad = old.m_rad;
 }
 ...
private:
  m_x      m_y      m_rad
}
```

a

```
class Circ
{
 ...
 Circ(float x, float y, float rad)
 {
   m_x = x;
   m_y = y;
   m_rad = rad;
 }
 ...
private:
  m_x  1   m_y  2   m_rad  10
}
```

```
void foo(Circ temp)
{
   cout << "Area is: "
        << temp.GetArea();
}

int main()
{
   Circ a(1,2,10);

   foo(a);
}
```

Copy Construction

```
class Circ
{
public:
  Circ(float x, float y, float r)
  {
    m_x = x; m_y = y; m_rad = r;
  }
  Circ(const Circ& old)
  {
    m_x = oldVar.m_x;
    m_y = oldVar.m_y;
    m_rad = oldVar.m_rad;
  }
  float GetArea()
  {
    return(3.14159*m_rad*m_rad);
  }
private:
  float m_x, m_y, m_rad;
};
```

- If you don't define your own copy constructor…
- C++ will provide a default one for you…
  It just copies all of the member variables from the old instance to the new instance…
- It's a byte-for-byte copy of the data!
- This is called a "shallow copy."
- But then why would I ever need to define my own copy constructor?
- Let's see!

```
int main()
{
    Circ a(1,2,3);

    Circ b(a);
}
```

Copy Construction

Ok – so why would we ever need to write our own Copy Constructor function?

After all, C++ shallow copies all of the member variables for us automatically if we don't write our own!

Well, we'll see very soon.

But first, let's go back to our PiNerd class…

The PiNerd Class

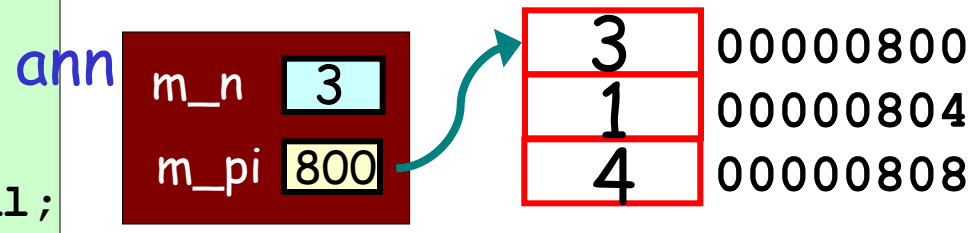
```
class PiNerd
{
public:
  PiNerd(int n) {
    m_n = n;
    m_pi = new int[n];
    for (int j=0;j<n;j++)
      m_pi[j] = getPiDigit(j);
  }

  ~PiNerd(){delete []m_pi;}

  void showOff()
  {
    for (int j=0;j<m_n;j++)
      cout << m_pi[j] << endl;
   }
private:
    int *m_pi, m_n;
};
```

- As you recall, every PiNerd memorizes the first N digits of π.
- Also recall that PiNerd uses new and delete to dynamically allocate memory for its array of N digits.
  - When constructed, it uses new to dynamically allocate an array to hold the first N digits of π.
  - And when it is destructed, it uses delete [] to release this array.
- Let's see what happens when we use this class in a simple program.

Copy Construction

```
class PiNerd
{
public:
  PiNerd(int n) {
    m_n = n;
    m_pi = new int[n];
    for (int j=0;j<n;j++)
      m_pi[j] = getPiDigit(j);
  }

  ~PiNerd(){delete []m_pi;}

  void showOff()
  {
    for (int j=0;j<m_n;j++)
      cout << m_pi[j] << endl;
   }
private:
    int *m_pi, m_n;
};
```

```
int main()
{
  PiNerd ann(3);
  if (...)
  {
     PiNerd ben = ann;
     ...
  }

  ann.showOff();
}
```



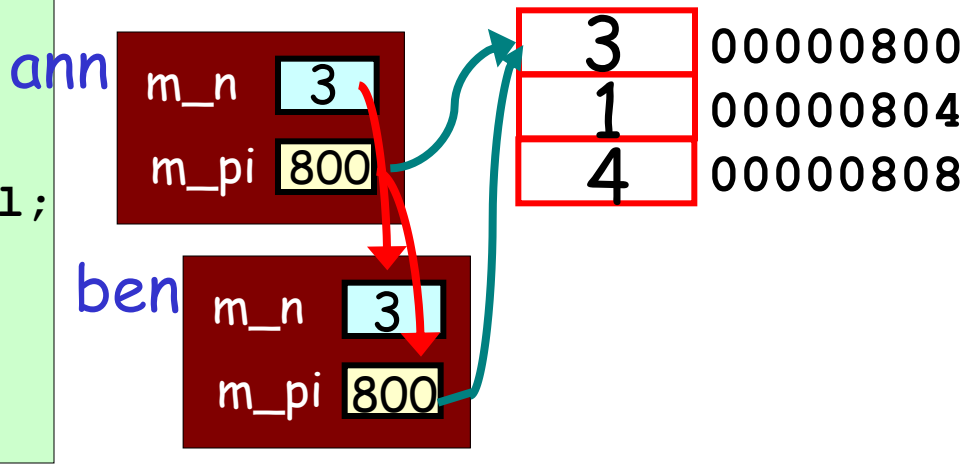
- In the above program, we construct ann and she knows 3 digits of Pi. See above for a diagram.

Copy Construction

- Because PiNerd doesn't have a explicit copy constructor, C++ simply copies every member variable from ann into the new ben variable (#1). You can see the member variables are identical after line #1.
- Notice that after the copy was made, ann and ben both point to the same array of 3 pi digits at location 800.
- That's ann's original array. It should belong JUST to ann, but now ben points at that same array too!

```
class PiNerd
{
public:
  PiNerd(int n) {
    m_n = n;
    m_pi = new int[n];
    for (int j=0;j<n;j++)
      m_pi[j] = getPiDigit(j);
  }

  ~PiNerd(){delete []m_pi;}

  void showOff()
  {
    for (int j=0;j<m_n;j++)
      cout << m_pi[j] << endl;
   }
private:
    int *m_pi, m_n;
};
```

```
int main()
{
  PiNerd ann(3);
  if (...)
  {
    PiNerd ben = ann; // #1
      ...
  }

  ann.showOff();
}
```

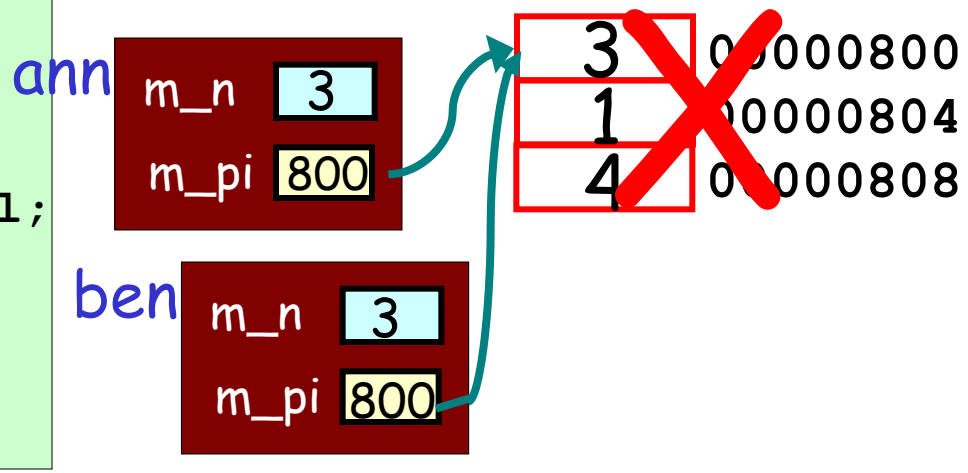


Copy Construction

- It's a problem that both ann and ben point to the same array! Why? Because when we hit line #1, ben's destructor is called because the ben variable goes out of scope.
- This causes ben's destructor to use the delete command to free the array that ben points to, at address 800.
- But because both ben and ann share the same array, now ben is deleting the array that ann owns!

```
class PiNerd
{
public:
  PiNerd(int n) {
    m_n = n;
    m_pi = new int[n];
    for (int j=0;j<n;j++)
      m_pi[j] = getPiDigit(j);
  }

  ~PiNerd(){delete []m_pi;}

  void showOff()
  {
    for (int j=0;j<m_n;j++)
      cout << m_pi[j] << endl;
   }
private:
    int *m_pi, m_n;
};
```

```
int main()
{
  PiNerd ann(3);
  if (...)
  {
      PiNerd ben = ann;
      ...
→ } // ben's d'tor called #1

  ann.showOff();
}
```

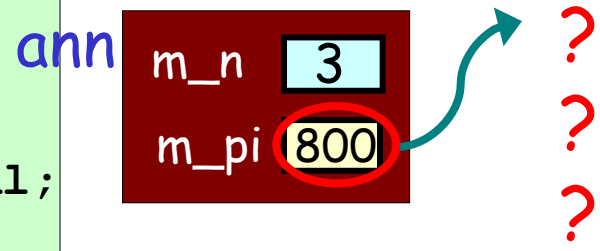


Copy Construction

- Later, when we try to use the ann variable, as in ann.showOff(), ann's m_pi member variable no longer points at a valid array! It still points at location 800, but this is no longer owned by ann since ben freed it!
- The memory at 800 might be used for something else now. It might have different values, etc.
- So ann's print function might print random values, or just crash. Or it might get lucky and work just fine sometimes, hiding a bug.

```
class PiNerd
{
public:
  PiNerd(int n) {
    m_n = n;
    m_pi = new int[n];
    for (int j=0;j<n;j++)
      m_pi[j] = getPiDigit(j);
  }

  ~PiNerd(){delete []m_pi;}

  void showOff()
  {
    for (int j=0;j<m_n;j++)
      cout << m_pi[j] << endl;
   }
private:
    int *m_pi, m_n;
};
```

```
int main()
{
  PiNerd ann(3);
  if (...)
  {
     PiNerd ben = ann;
     ...
  }// ben's d'tor called

→ann.showOff();
}
```



That's a big problem!

Copy Construction

- What's the moral of the story?
- Any time an object holds one or more pointer member variables and you make a shallow copy of the object…
- Bad things will happen when you destruct either the original variable or its copy(ies), since that will cause the other variable's pointer to point to invalid data!
- This is also true for many types of member variables, including pointers, file objects, network sockets, etc.

```
class PiNerd
{
public:
  PiNerd(int n) {
    m_n = n;
    m_pi = new int[n];
    for (int j=0;j<n;j++)
      m_pi[j] = getPiDigit(j);
  }

  ~PiNerd(){delete []m_pi;}

  void showOff()
  {
    for (int j=0;j<m_n;j++)
      cout << m_pi[j] << endl;
   }
private:
    int *m_pi, m_n;
};
```

```
int main()
{
  PiNerd ann(3);
  if (...)
  {
     PiNerd ben = ann;
     ...
  }// ben's d'tor called

  ann.showOff();
}
```

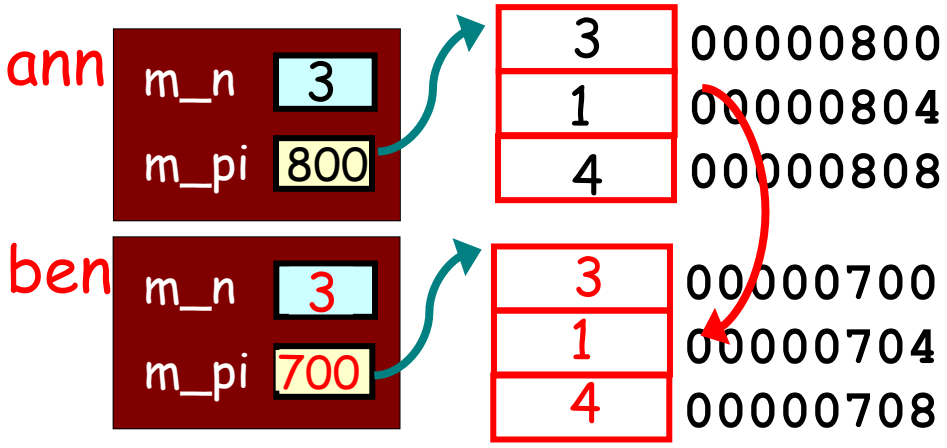
Copy Construction

So how do we fix this?

For such classes, you must define your own *copy constructor!*

Here's how it works for
PiNerd ben = ann;

1. Determine how much memory is allocated by the old variable.
2. Allocate the same amount of memory in the new variable.
3. Copy the contents of the old variable to the new variable.



The Copy Constructor

```
class PiNerd
{
public:
   PiNerd(int n) { ... }
   ~PiNerd(){ delete[]m_pi; }

   // copy constructor
   PiNerd(const PiNerd &src)
   {
     m_n = src.m_n;



   }

   void showOff() { ... }

private:
    int *m_pi, m_n;
};
```

```
int main()
{
   PiNerd ann(3);
   if (...)
   {
      PiNerd ben = ann;
      ...
   }

   ann.showOff();
```

This means:
"The new instance must have the same number of array slots as the old instance."

First our copy constructor must determine how much memory is required by the new instance.

Let's see how to define our copy constructor!

The Copy Constructor

```
class PiNerd
{
public:
   PiNerd(int n) { ... }
   ~PiNerd(){ delete[]m_pi; }

   // copy constructor
   PiNerd(const PiNerd &src)
   {
     m_n = src.m_n;
     m_pi = new int[m_n];


   }

   void showOff() { ... }

private:
    int *m_pi, m_n;
};
```

```
int main()
{
   PiNerd ann(3);
   if (...)
   {
       PiNerd ben = ann;
       ...
   }

   ann.showOff();
}
```

Next, our copy constructor must allocate its own copy of any dynamic memory!

This ensures that the new instance has its own array and doesn't share the old instance's array!

Let's see how to define our copy constructor!

The Copy Constructor

```
class PiNerd
{
public:
  PiNerd(int n) { ... }
  ~PiNerd(){ delete[]m_pi; }

  // copy constructor
  PiNerd(const PiNerd &src)
  {
    m_n = src.m_n;
    m_pi = new int[m_n];
    for (int j=0;j<m_n;j++)
      m_pi[j] = src.m_pi[j];
  }

  void showOff() {      }

private:
   int *m_pi, m_n;
};
```

```
int main()
{
  PiNerd ann(3);
  if (...)
  {
     PiNerd ben = ann;
     ...
  }

  ann.showOff();
}
```

Finally, we have to manually copy over the contents of the original array to our new array.

This ensures that the new instance has its own copy of all of the array data!

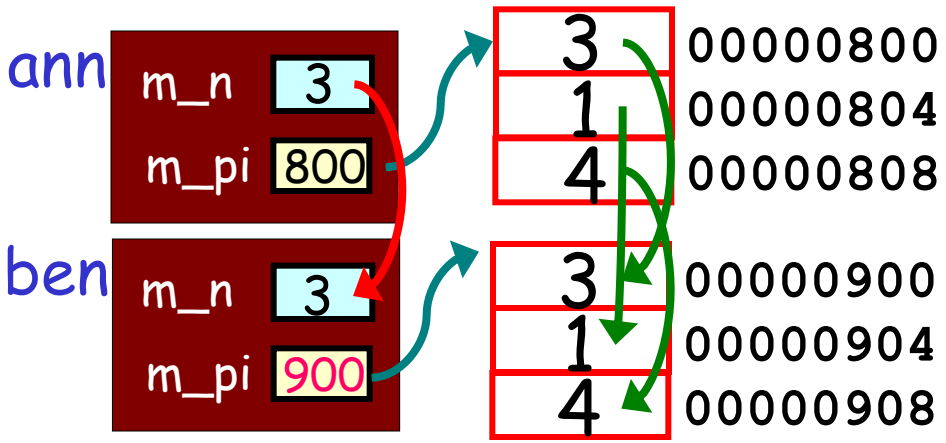
Let's see how to define our copy constructor!

The Copy Constructor

```
class PiNerd
{
public:
   PiNerd(int n) { ... }
   ~PiNerd(){ delete[]m_pi; }

   // copy constructor
   PiNerd(const PiNerd &src)
   {
     m_n = src.m_n;
     m_pi = new int[m_n];
     for (int j=0;j<m_n;j++)
       m_pi[j] = src.m_pi[j];
   }

   void showOff() { ... }

private:
    int *m_pi, m_n;
};
```

```
int main()
{
   PiNerd ann(3);
   if (...)
   {
     PiNerd ben = ann;
     ...
   }

   ann.showOff();
}
```

- Now when our ben variable is created, it allocates its own dynamic array (at address 900) and copies ann's values over into its array
- ben and ann now each have their own distinct array. The arrays hold identical values (3,1,4) but at different locations



The Copy Constructor

```
class PiNerd
{
public:
   PiNerd(int n) { ... }
   ~PiNerd(){ delete[]m_pi; }

   // copy constructor
   PiNerd(const PiNerd &src)
   {
     m_n = src.m_n;
     m_pi = new int[m_n];
     for (int j=0;j<m_n;j++)
       m_pi[j] = src.m_pi[j];
   }

   void showOff() { ... }

private:
    int *m_pi, m_n;
};
```

We're A-OK, since ann still has its own array!

```
int main()
{
   PiNerd ann(3);
   if (...)
   {
      PiNerd ben = ann;
      ...
→} // ben's d'tor called #1

→ann.showOff(); // #2
}
```

3
1
4

- When we hit line #1, ben's destructor runs and deletes ben's array at location 900
- At the end of the destructor, the ben variable disappears too
- But notice that ann is just fine, since her variable has a separate array at 800
- So the printout on line #2 works great