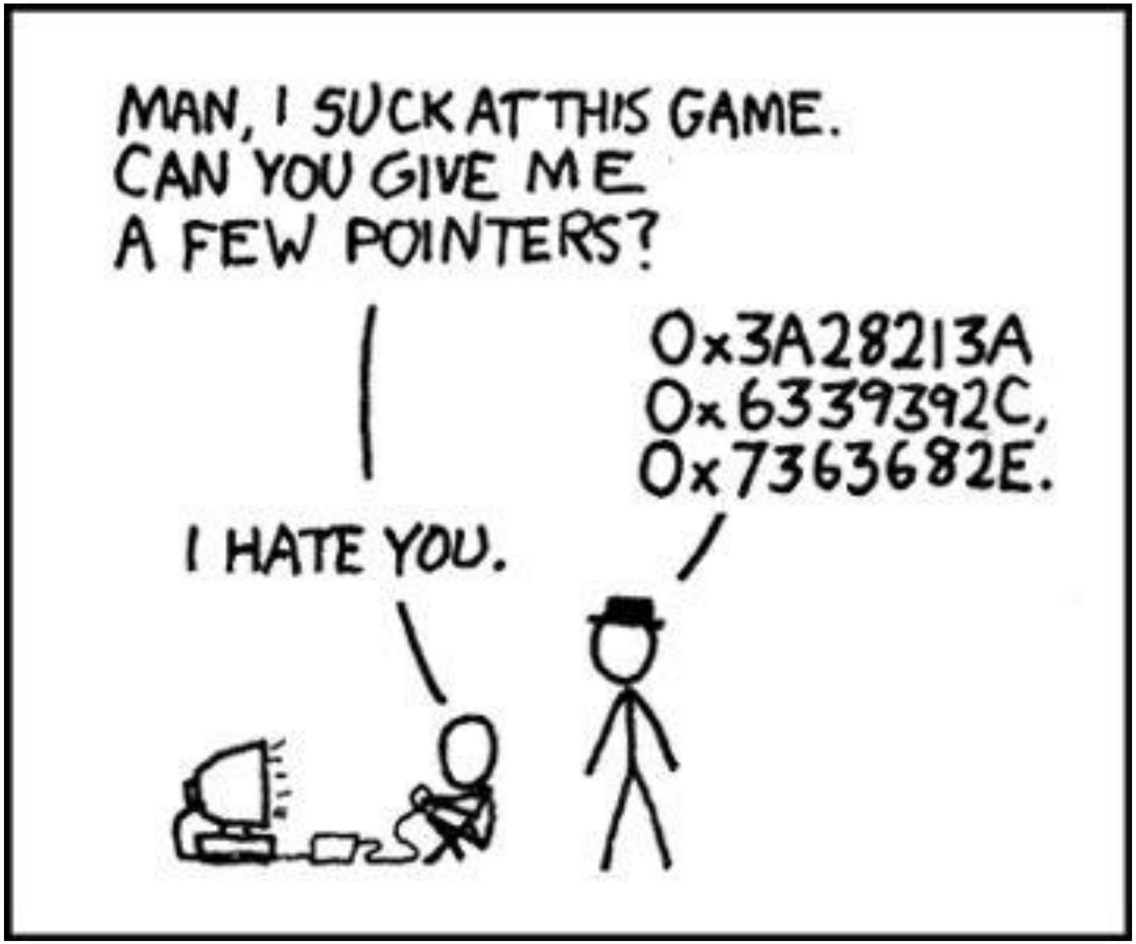
# Lecture #3

- 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.

# Pointers…
# Why should you care?


Pointers are a critical feature of C and C++.

And you'll struggle during the rest of CS32 if you don't understand them super well.

And job interviewers love asking pointer questions.

So pay attention!

# Every Variable Has An Address

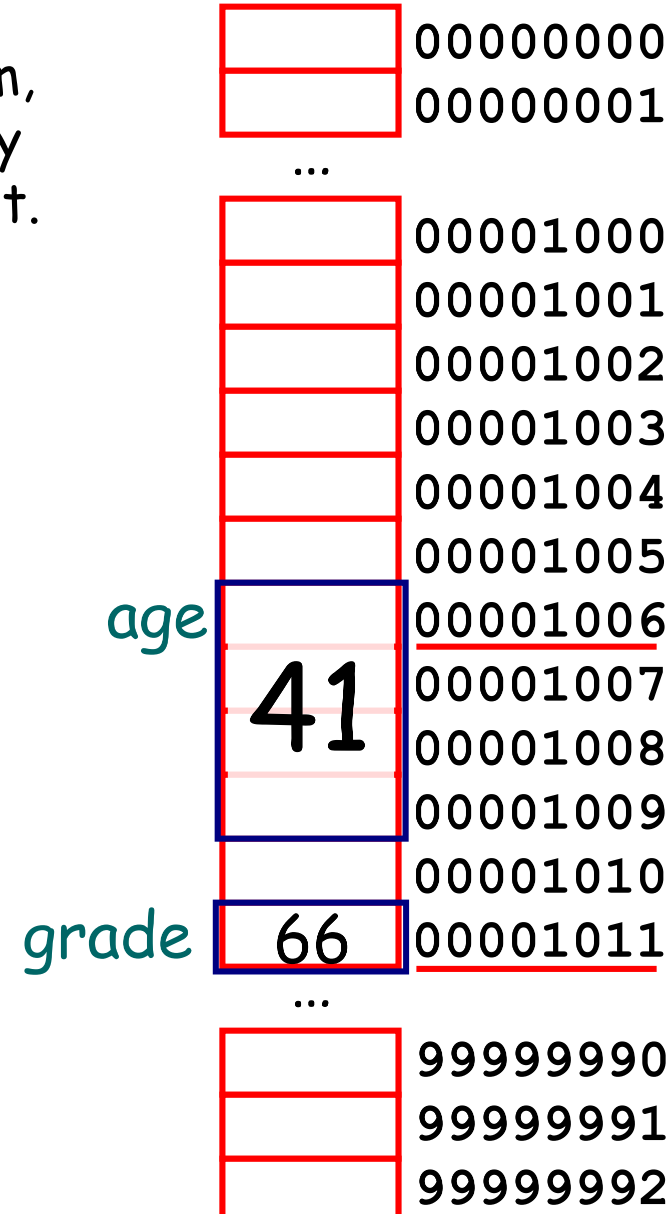
Every time you define a variable in your program, the compiler finds an unused address in memory and reserves one or more bytes there to store it.

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

So, what is age's address in memory?

What about grade's address?

```
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.

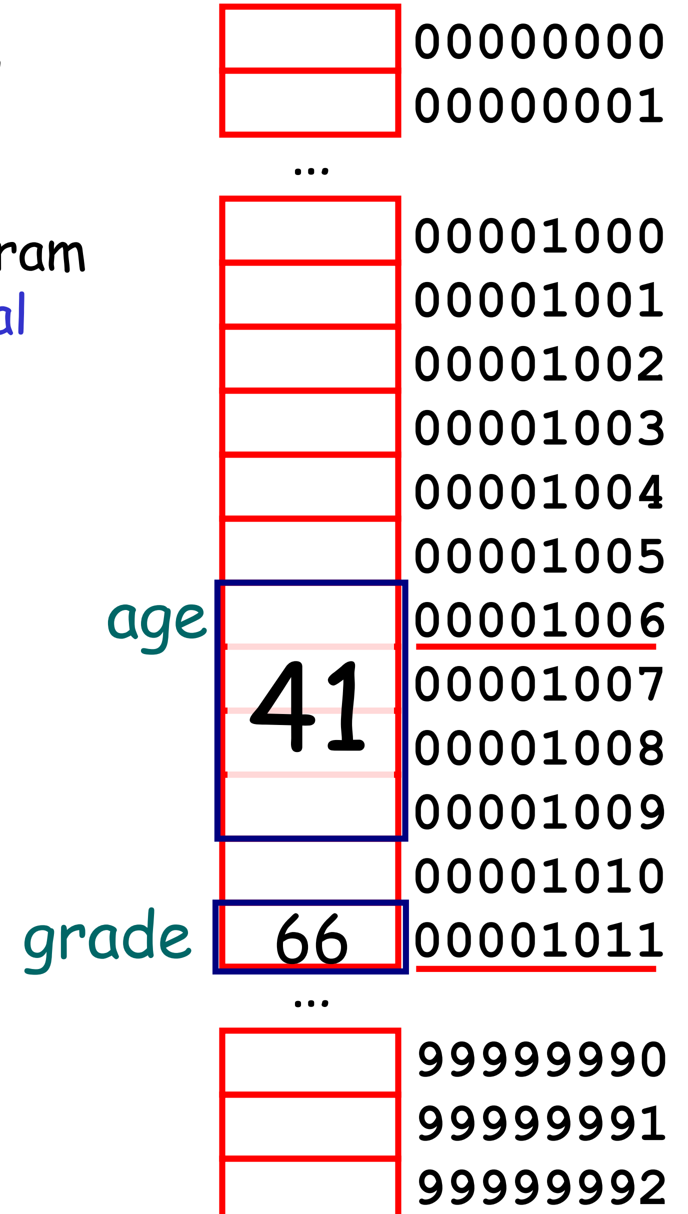
If you place an & before a variable in a program statement, it means "give me the numerical address of the variable."

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?

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

I'm a regular variable… and I hold a regular value!


| | | |
|---|---|---|
| idontknow | 5 | 00001000 |
| | | 00001002 |
| | | 00001004 |
| | | 00001006 |
| | | 00001008 |
| p | 1000 | 00001010 |
| | | 00001012 |
| | | 00001014 |
| | | . . . |

Another way to say this is: "p points to address 1000"

```
int foo()
{
    int idontknow;

    idontknow = 5;

    int *p;

    p = &idontknow;
}
```

I'm a pointer variable… and all I can hold are the addresses of other variables!

To understand the type of your pointer variable, simply read your declaration from right to left…

# 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.

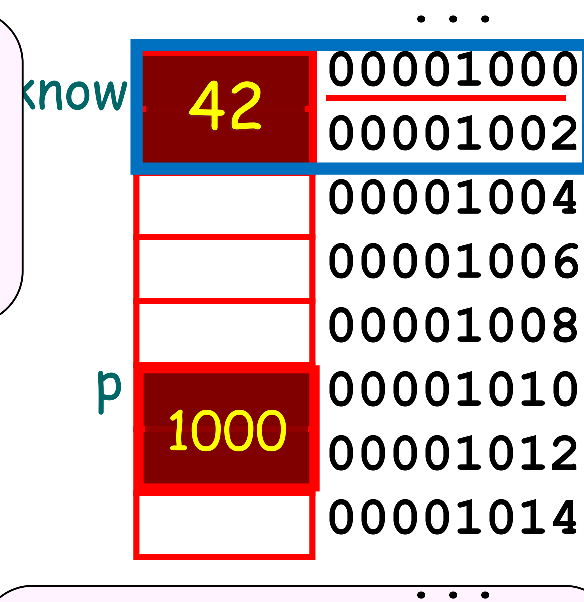
If placed in front of a pointer, the * operator allows us to read/write the variable pointed-to by the pointer.

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

   int *p;
   p = &idontknow;

   cout << *ptr;     // cout << *p → cout << *1000 → cout << 42
   *ptr   =  5;      // *p = 5 → *1000 = 5
}
```

"Get the address value stored in the ptr variable…

Then go to that address in memory…

and give me the value stored there."

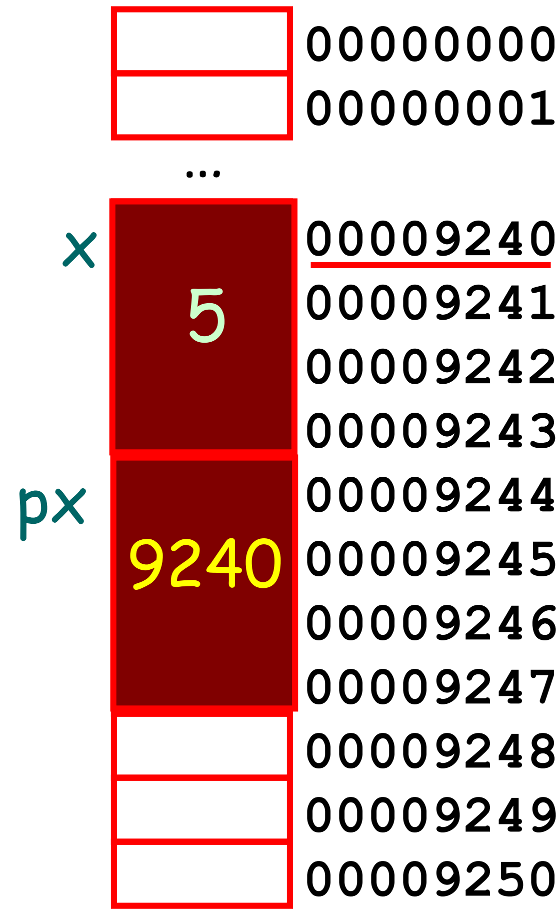



"Get the address value stored in the p variable… Then go to that address in memory… and store a value of 5 there."

# Another Pointer Example

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

int main()
{
   int x = 1;

   set(&x);

   cout << x; // prints 5
}
```

"Store a value of 5 at location 9240."



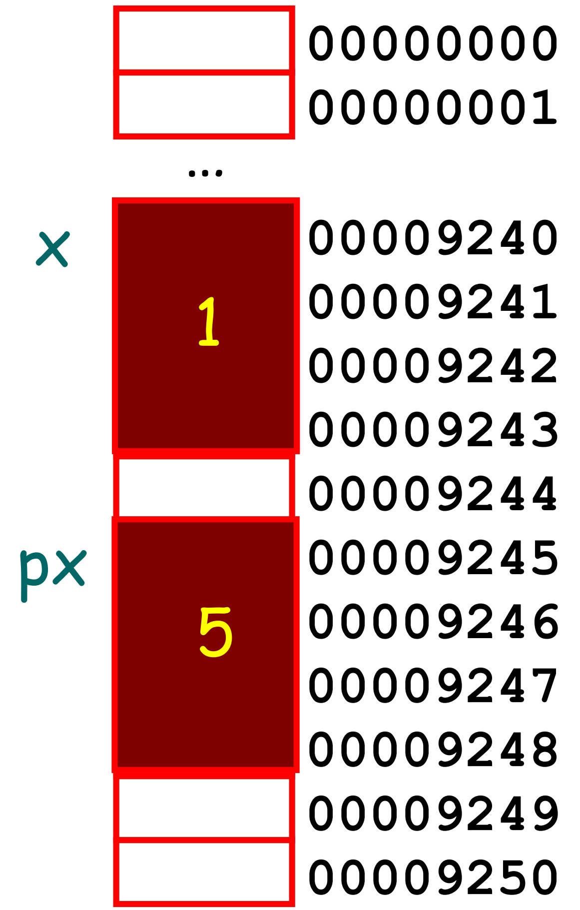

Let's use pointers to modify a variable inside of another function.

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

# What if We Didn't Use Pointers?

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

int main()
{
   int x = 1;

   set(x);

   cout << x; // prints 1
}
```

Now what would happen if we didn't use pointers in our code?




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

Had we used a pointer, it would have worked!

# Pointers vs Refere

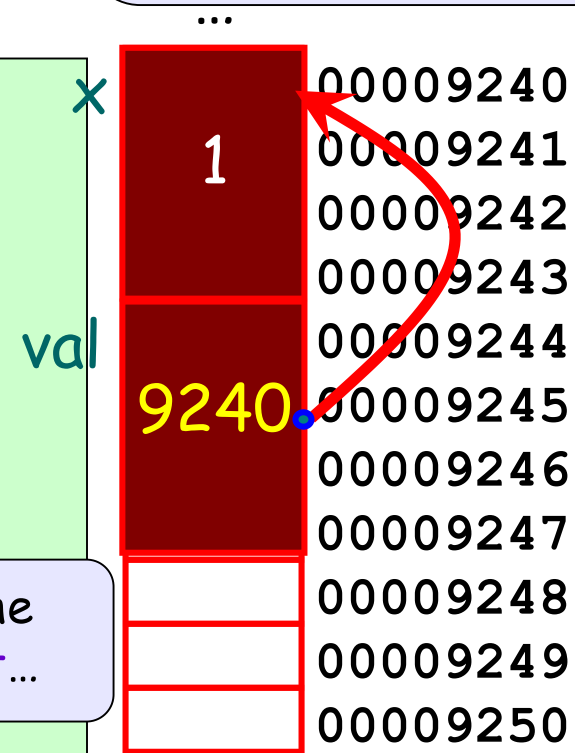
Since val points to our original variable, x, this line actually changes x!

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

```
void set(int &val) // val is a ref
{
   val = 5;
}

int main()
{
   int x = 1;

   set(x);
   cout << x;
}
```

It looks like we're just passing the value of x, but in fact…

This line is really passing the address of variable x to set…




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

(Under the hood, C++ uses a pointer)

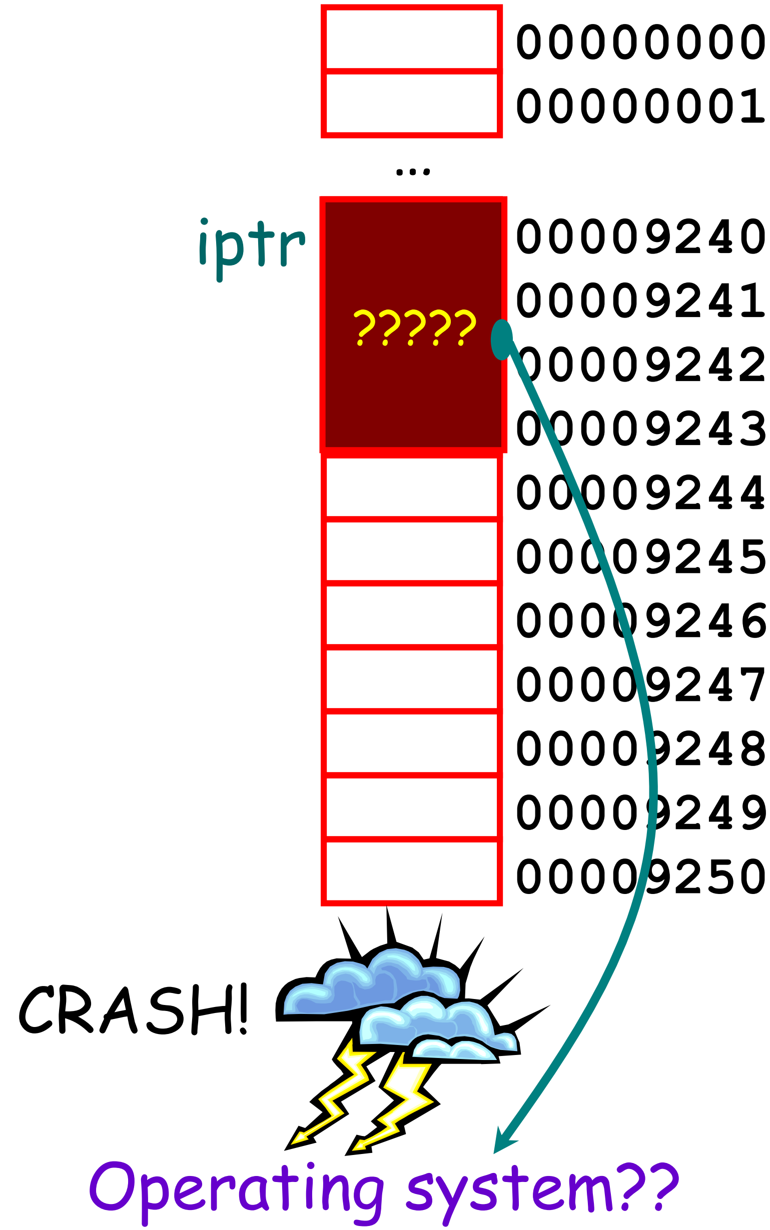
# What Happens Here?

```
int main()
{
   int *iptr;

   *iptr = 123456; // #1 mistake!
}
```

What address does iptr hold?

Who knows??? Since the programmer didn't initialize it, it points to some random spot in memory!

Moral:
You must always set the value of a pointer variable before using the * operator on it!



# Class Challenge

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

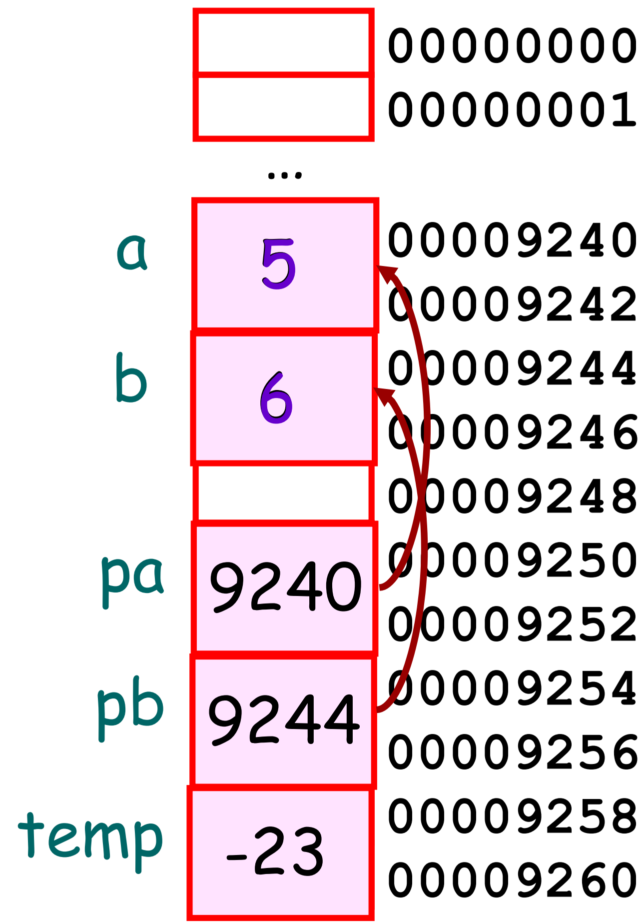
Prize: 3 prize tickets (and maybe some candy)

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

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

Hint: Make sure you never use a pointer unless you point it to a variable first!!!

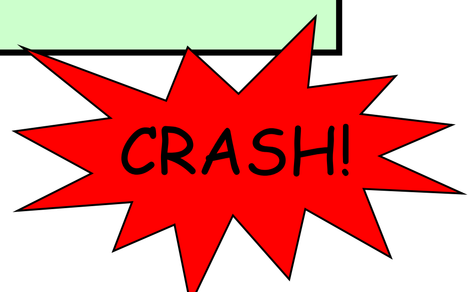
# Class Challenge Solution

```
void swap (int *pa, int *pb)
{
  int temp;
  temp = *pa;
  *pa = *pb;
  *pb = temp;
}

int main()
{
   int a=5, b=6;
       9240, 9244
   swap(&a,&b);
   cout << a;
   cout << b;
}
```

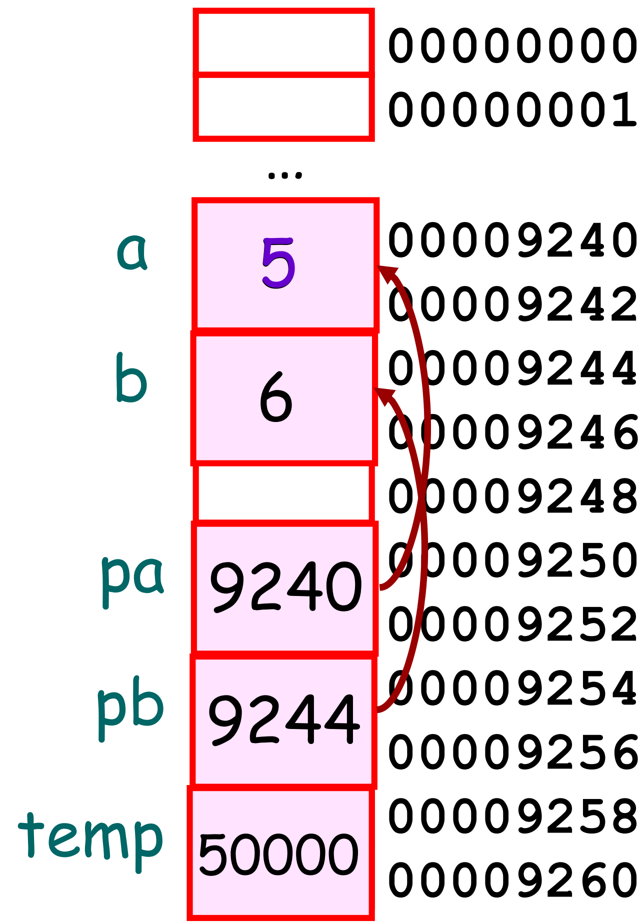


# Wrong Challenge Solution #1

```
void swap (int *pa, int *pb)
{
   int *temp;
   *temp = *pa;
   *pa = *pb;
   *pb = *temp;
}

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

    swap(&a,&b);
    cout << a;
    cout << b;
}
```



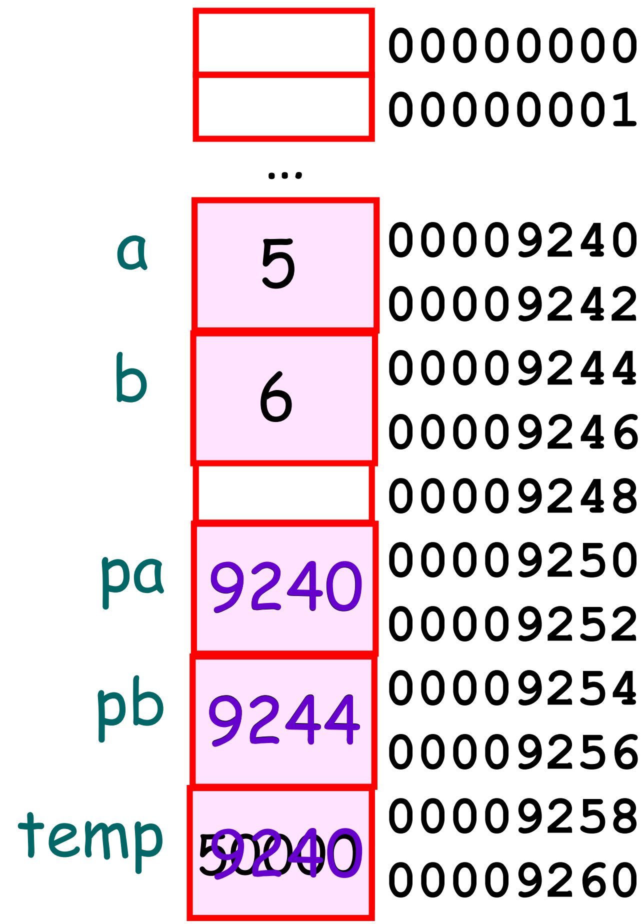

Problem: In this solution, we use a pointer without first pointing it at a variable!



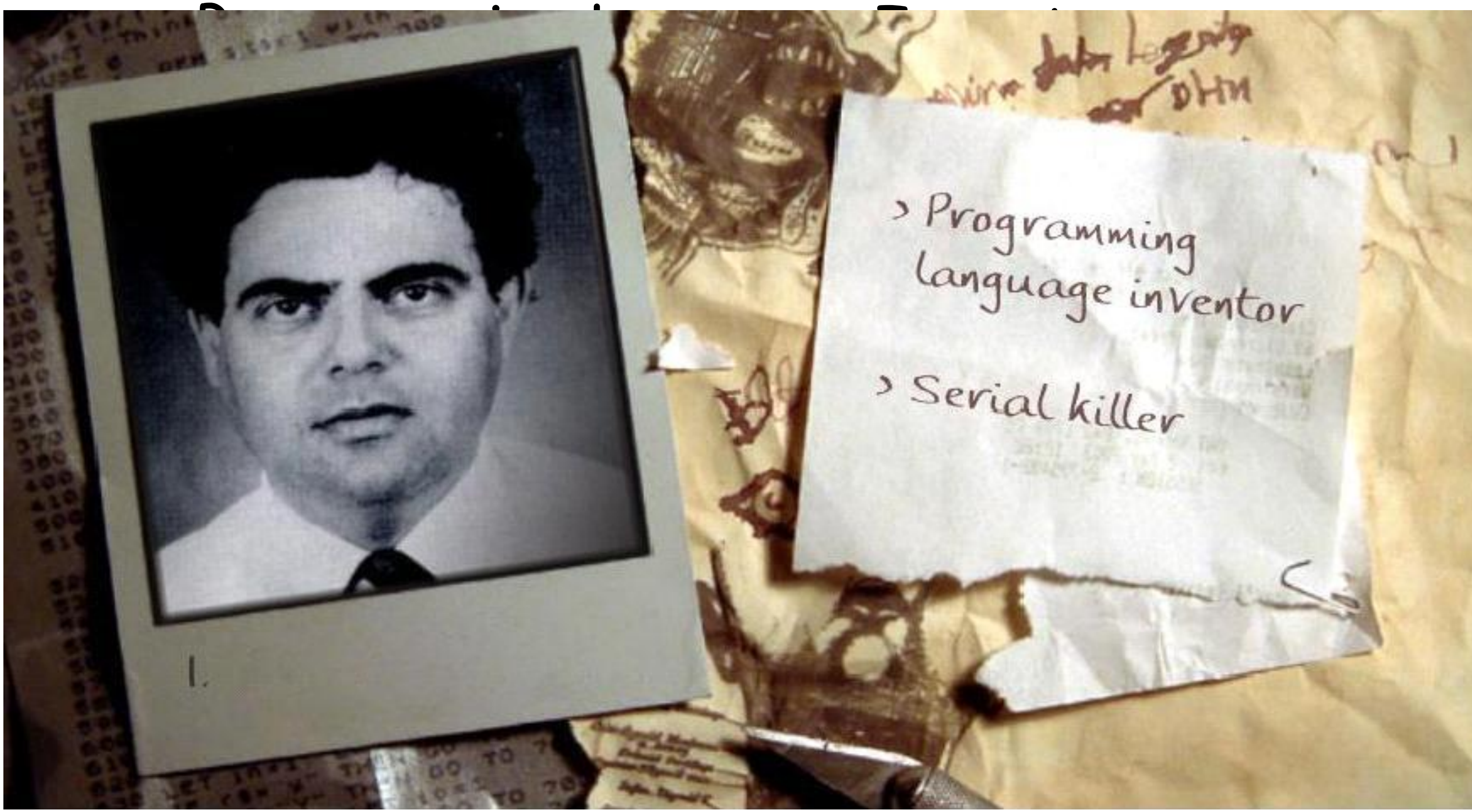
# Wrong Challenge Solution #2

```
void swap (int *pa, int *pb)
{
   int *temp;
   temp = pa;
   pa = pb;
   pb = temp;
}

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

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

Problem: In this solution, we swap the pointers but not the values they point to!



# Let's Play....



# Arrays, Addresses and Pointers

Just like any other variable, every array has an address in memory.

But… 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!

And here's how to make a pointer point to an array…

Question: So is "nums" an address or a pointer or what?

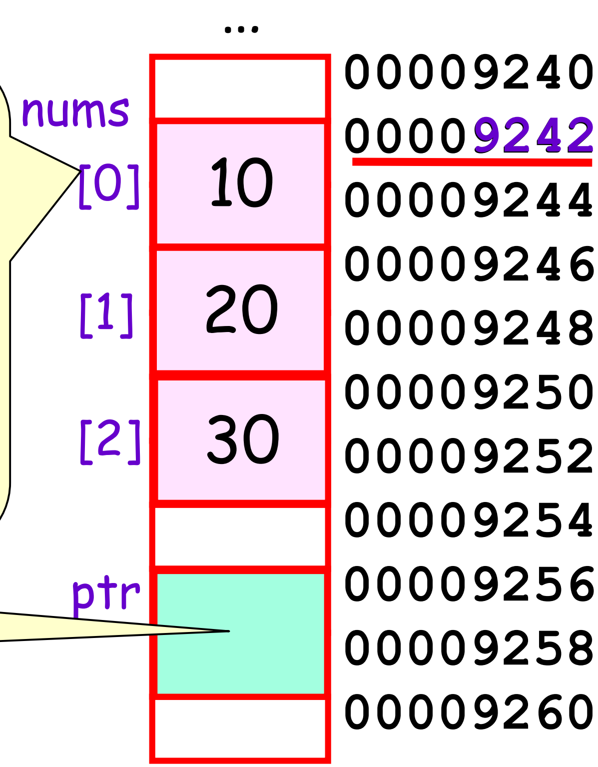
Answer: "nums" is just an array. But C++ lets you get its address without using the & so it looks like a pointer…

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

   cout <<   nums;  // prints 9242

   int *ptr = nums;   // pointer to array

}
```

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!

ptr is a pointer variable.

Why? Because it's a variable that holds an address value!



# 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.

NOTE: when we say "skip down j elements," we don't just mean "skip down j bytes!"

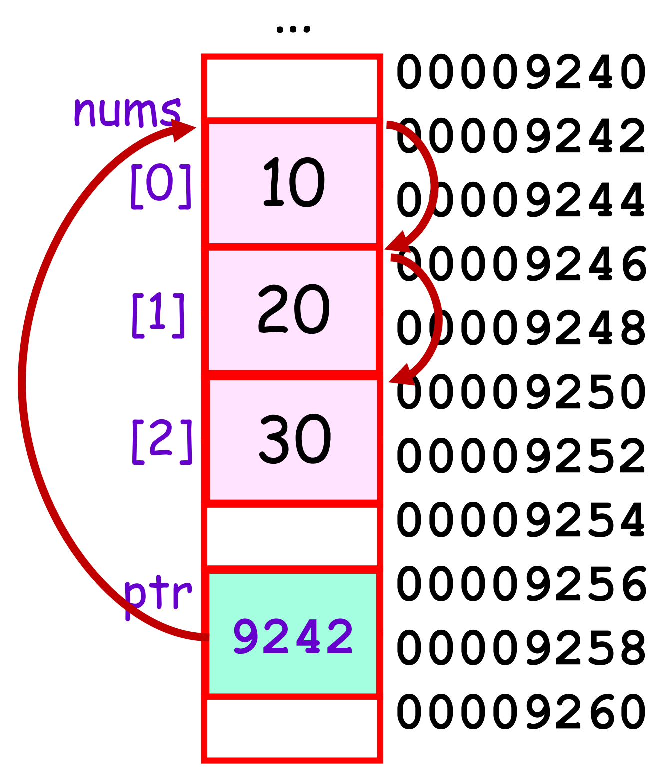
Instead we mean, skip over j of the actual elements/values in the array (e.g., skip over the values 10 and 20 to get to 30)

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

   int *ptr = nums;  // pointer to array

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

}
```

In C++, the two syntaxes have identical behavior:

ptr[j] ↔ *(ptr + j)

They both mean: "Go to the address in ptr, then skip down j elements and get the value."



# Pointer Arithmetic and Arrays

The array parameter variable is actually a pointer!

You can use [ ] syntax if you like but it's REALLY a pointer!

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 array itself!

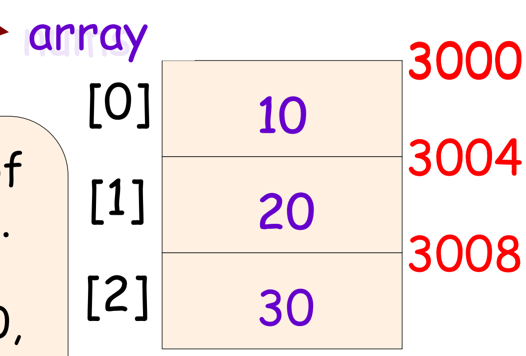
array

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

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

   printData(nums);

   printData(&nums[1]);

   printData(nums+1);
}
```

Here we're passing the address of the second element of the array.

Since nums[0] is at address 3000, nums[1] one is 4 bytes down at 3004.



# Pointer Arithmetic and Arrays

When you use recursion on arrays, you'll often use this notation…

To process successively smaller suffixes of the array.

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

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

   printData(nums);

   printData(&nums[1]);

   printData(nums+1);
}
```

This line is tricky! First, what happens when you just write the name of an array all by itself?
Answer: C++ replaces the name with the start address of the array.

nums

| | | |
|---|---|---|
| [0] | 10 | 3000 |
| [0] | 20 | 3004 |
| [1] | 30 | 3008 |

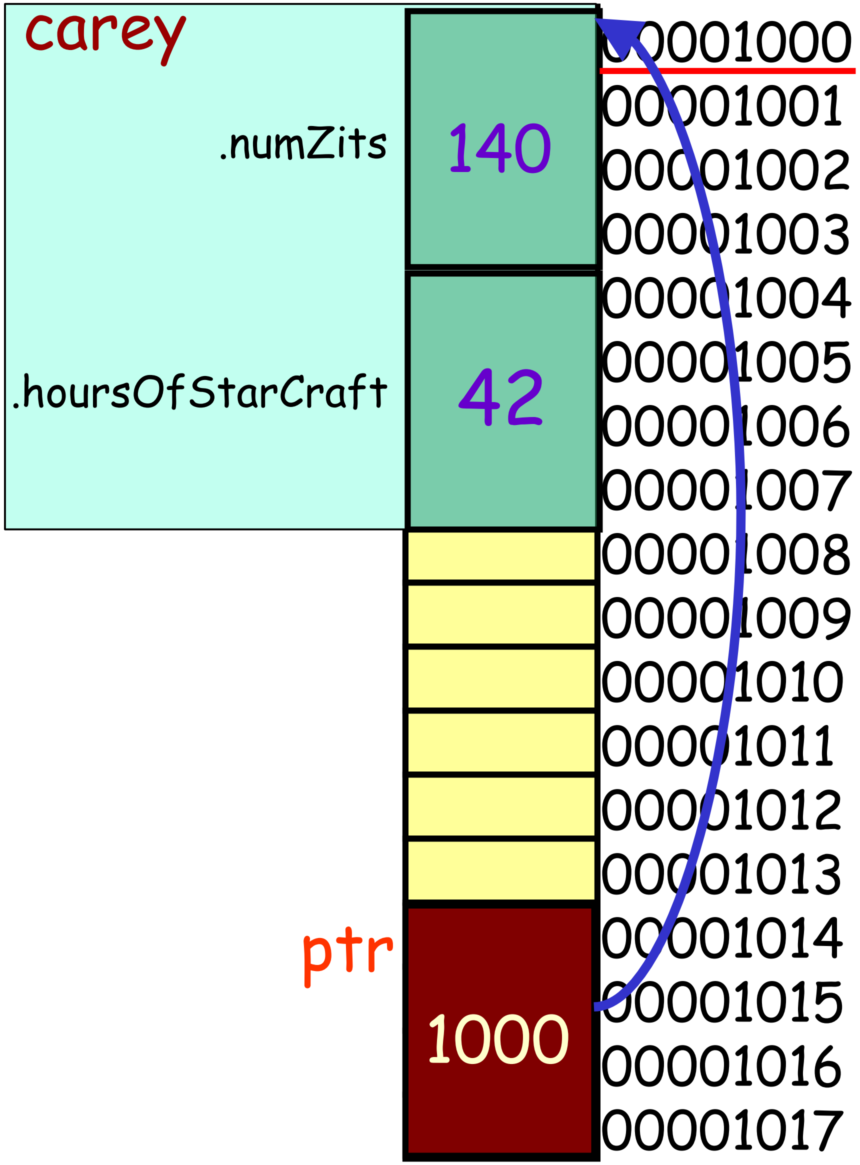
Next, we add 1 to this address:
nums + 1
This means:
"Advance one element (one integer) down from the start of the nums array."

# 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(void)
      {  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.

The area is:  314

foo

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

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

   ...

private:
  m_x [   ]   m_y [   ]   m_rad [   ]
};
```

3000
3001
3002
3003
3004
3005
3006
3007
3008
3009
3010
...

```
void printInfo(Circ *ptr)
{
   cout << "The area is: ";
   cout << ptr->getArea();
}

int main()
{
   Circ foo(3,4,10);

   printInfo(&foo);
}
```

ptr 3000

# Classes and the "this" Pointer

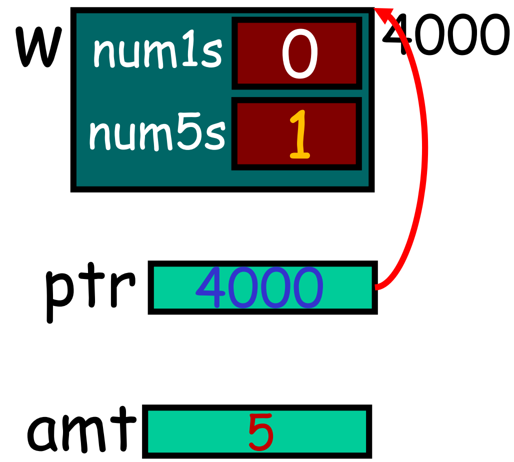
Before C++, in the dark ages when Carey learned programming, we didn't use classes!

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(void)
{
  Wallet w;

  Init(&w);

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

As it turns out, C++ classes work in an almost identical fashion!

# 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…

And here's how we might use our class…

```
int main()
{
  Wallet a;

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

# Classes and the "this" Pointer

Here what your Init() method looks like…

But here's what's REALLY happening! ☺

And C++ does the same thing to your actual member functions!

It adds a hidden first argument that's a pointer to your original variable!

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

```
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++;
}
...
```

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

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

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

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

# Classes and the "this" Pointer

C++ converts all of your member functions automatically and invisibly by adding an extra pointer parameter called "this":

Yes… the pointer is actually called "this"!

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

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

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

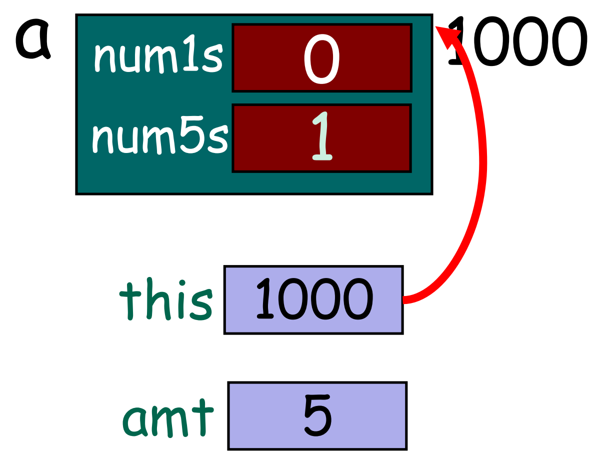
```
void Init(Wallet *this)
{
  this->num1s = this->num5s = 0;
}
void AddBill(Wallet *this, int amt)
{
   if (amt == 1)     this->num1s++;
   else if (amt==5) this->num5s++;
}
...
```

```
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;

  a.Init(&a);

  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!

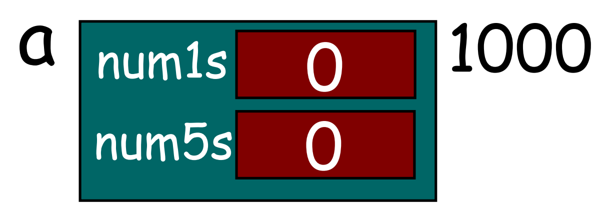
```
void Wal      ::Init()
{
    this->num1s = this->num5s = 0;
    cout << "I am at address: " << this;
}
void Wallet::AddBill(int amt)
{
    if (amt == 1)               num1s++;
    else if (amt == 5)          num5s++;
}
...
```

```
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!

So now you know how C++ classes work under the hood!




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 main()
3600 {
3650    FuncPtr f;
3700
3750    f = &squared;
3800    f(10);
3850    f = &cubed;
3900    f(2);
3950 }
```

f

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…

This line gets the address of our squared() function and puts it into f.

# 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 main()
3600 {
3650     void (*f)(int);
3700
3750     f = &squared;
3800     f(10);
3850     f = &cubed;
3900     f(2);
3950 }
```

Oh, and you can leave out the & if you like.

It works the same way!

YES! Just as you can have pointers to variables, in C++ you can also have pointers to functions!

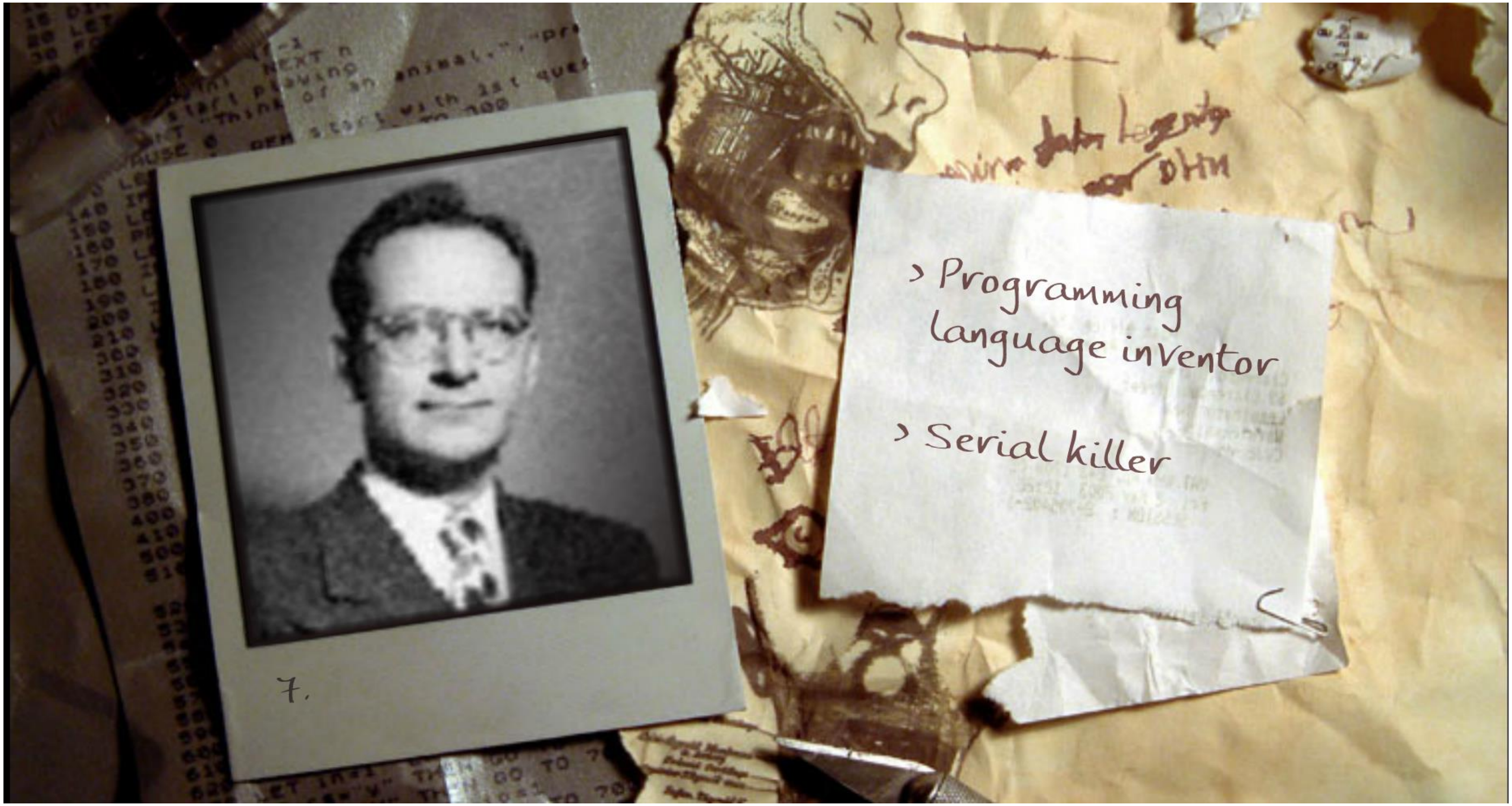
Let's gloss over the syntax for a second, and just see how it might work…

And here's the real syntax to declare a function pointer…

Don't worry about the syntax right now…

Just remember the concept.

# And now it's time for your favorite game!



# A New Type of Variable

Thus far, all variables we've defined have either been local variables, global variables or class member variables.

Let's learn about a new type of variable: a dynamic variable

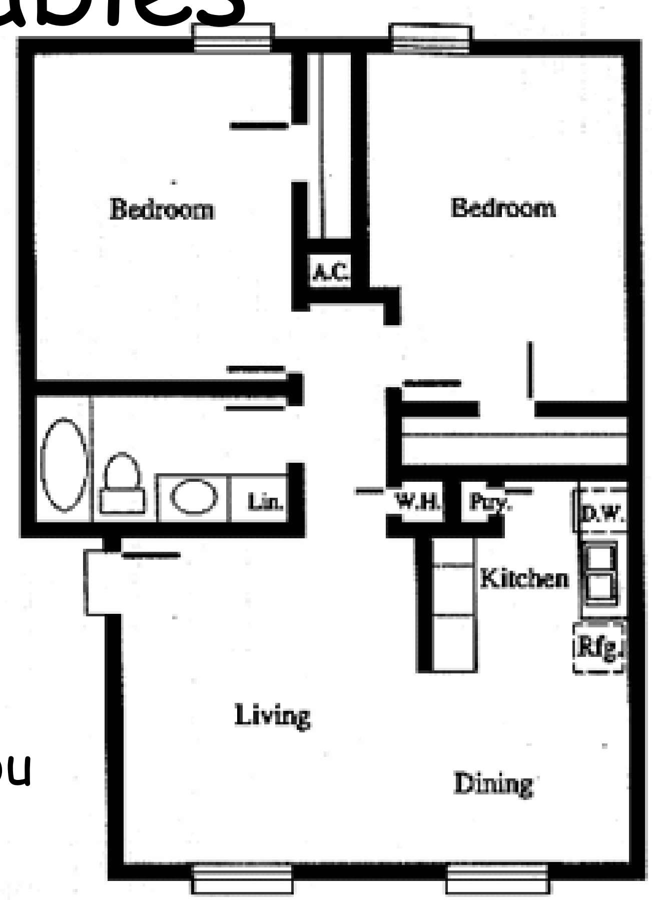
```
void foo(void)
{
  int a;
  cin >> a;
}

int aGlobalVariable;

int main()
{
  Circ a(3,4,10);
  float c[10];

  c[0] = a.getArea();
}
```

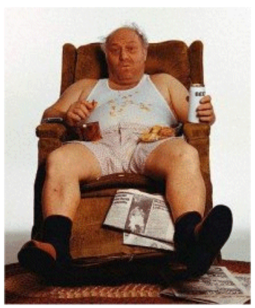
```
class Student
{
public:
  string getZits(void)
  {
    int numZits = m_age * 5;
    return(numZits);
  }
private:
  string m_name;
  int m_age;
};
```

# Dynamic Variables

You can think of traditional variables like rooms in your house.

Just like a room can hold a person, a variable holds a value.



But what if you run out of rooms because all of your aunts and uncles surprise you and come over.

In this case, you have to call a hotel, reserve some rooms, and place your relatives in the hotel rooms instead.

# Dynamic Variables

In a similar fashion, sometimes you won't know how many variables you'll need until your program runs.

In this case, you can dynamically ask the operating system to reserve new memory for variables.

The operating system will allocate room for your variable in the computer's free memory and then return the address of the new variable.

When you're done with the variable, you can tell the operating system to free the space it occupies for someone else to use.

# New and Delete

For example, 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(void)
{
    int *arr;          #1
    int size;

    cin >> size;       #2

    arr = new int[size];   #3

    arr[0] = 10;       #4
    arr[2] = 75;

    delete [] arr;     #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 just use the pointer just like it's an array!
5. Free the memory when you're done (check your relatives out of the hotel).

# New and Delete

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

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

    arr = new int[size];

    arr[0] = 1
    // 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.

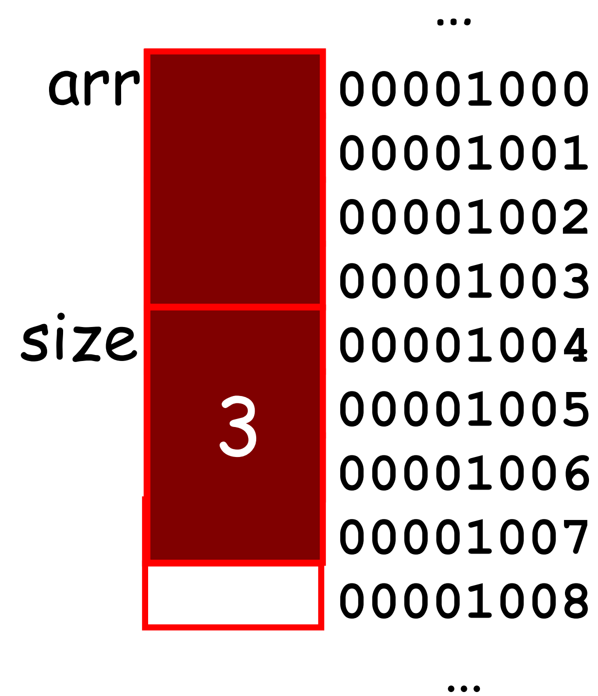
Make sure that the pointer's type is the same as the type of array you're creating!

# New and Delete

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

   cin >> size;

   arr = new int[size];

   arr[0] = 10;
   // etc

   delete [] arr;
}
```

3

4 * 3 = 12 bytes

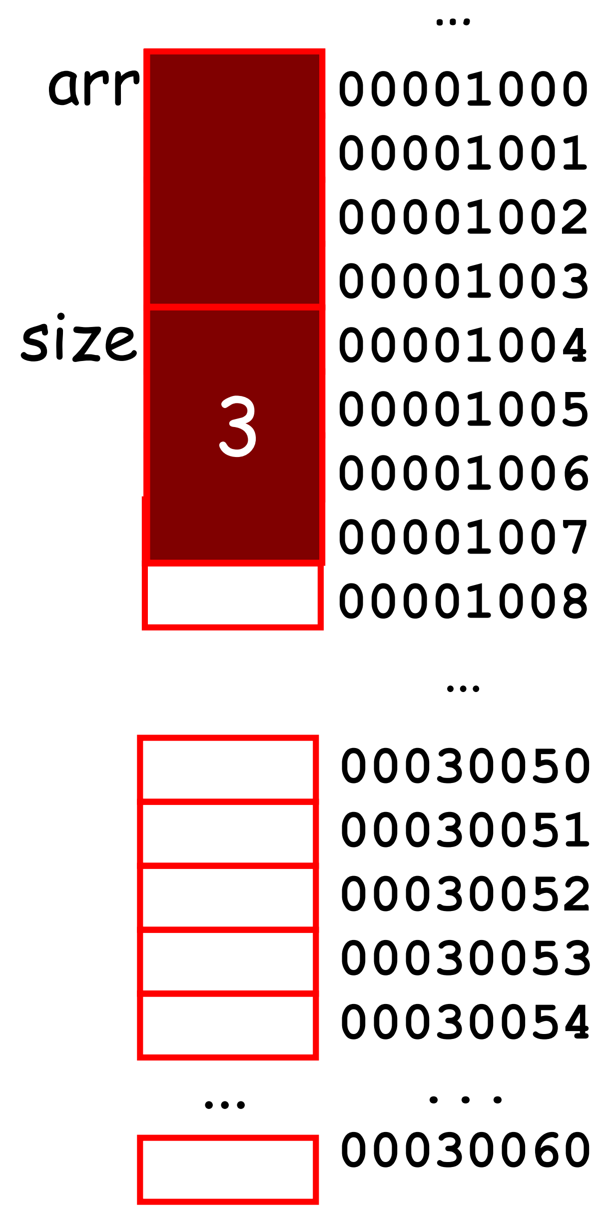



First, the new command determines how much memory it needs for the array.

# New and Delete

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

    cin >> size;

    arr = new int[size];

    arr[0] = 10;
    // etc

    delete [] arr;
}
```

4 * 3 = 12 bytes



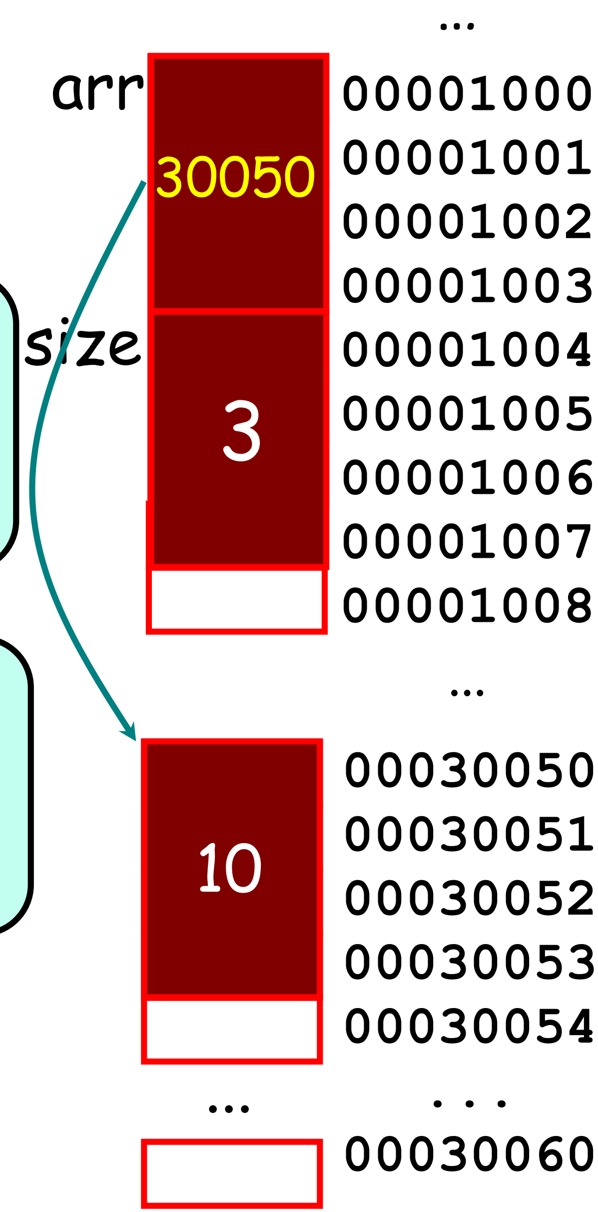

Next, the new command asks the operating system to reserve that many bytes of memory.

# New and Delete

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

    cin >> size;

                4  * 3   = 12
    arr = new int[size];

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

    delete [] arr;
}
```

You can also use the * notation if you like (instead of brackets)

You can now treat your pointer just like an array!
(i.e. use [ ] to index it)




Finally, your pointer variable gets the address of the newly reserved memory.

# ete

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

   cin >> size;

   arr = new int[size];

   arr[0] = 10;
   // etc

   delete [] arr;
   arr[0] = 50;
}
```

CRASH!

… not the pointer variable itself!

Our pointer variable still holds the address of the previously-reserved memory slots!

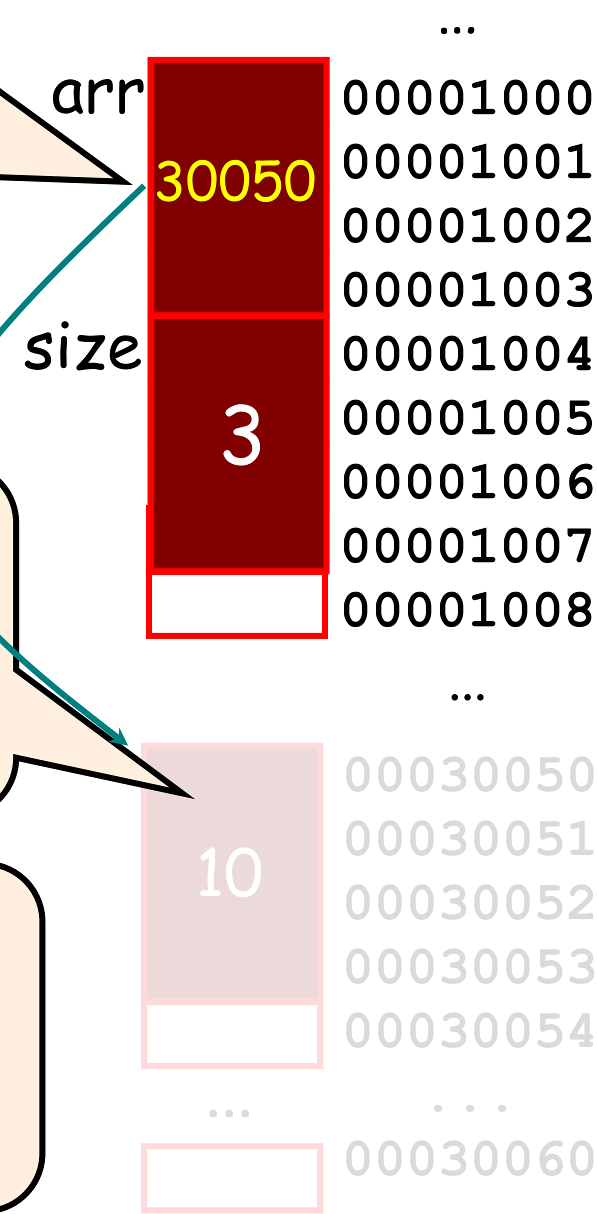
Note: When you use the delete command, you free the pointed-to memory…

But they're no longer reserved for this program!

So don't try to access them or bad things will happen!

When you're done, you use the delete command to free the array.

Usage: delete [] ptrname;



# New and Delete (For Non-Arrays)

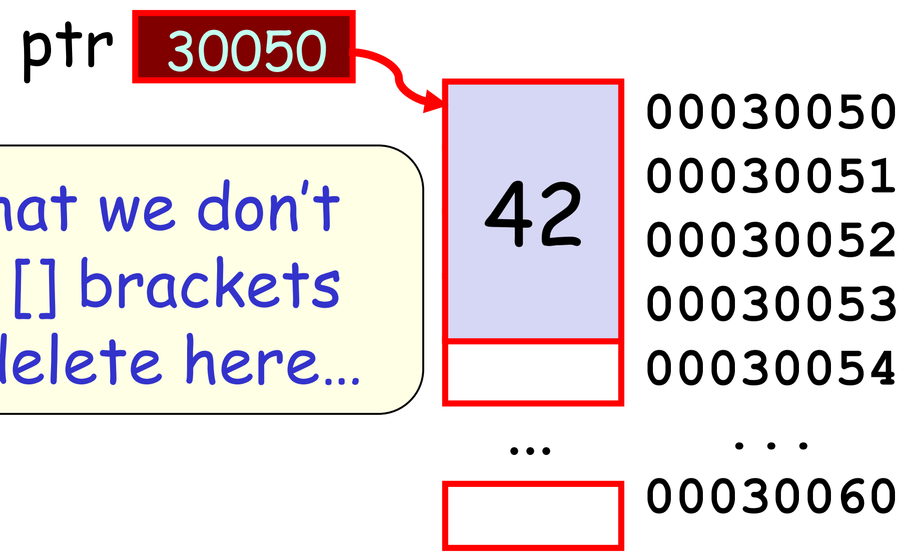
We can also use new and delete to dynamically create other types of variables as well!

For instance, we can allocate an integer variable like this…

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

    // allocate our dynamic variable
    ptr = new int;

    // use our dynamic var
    *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 (

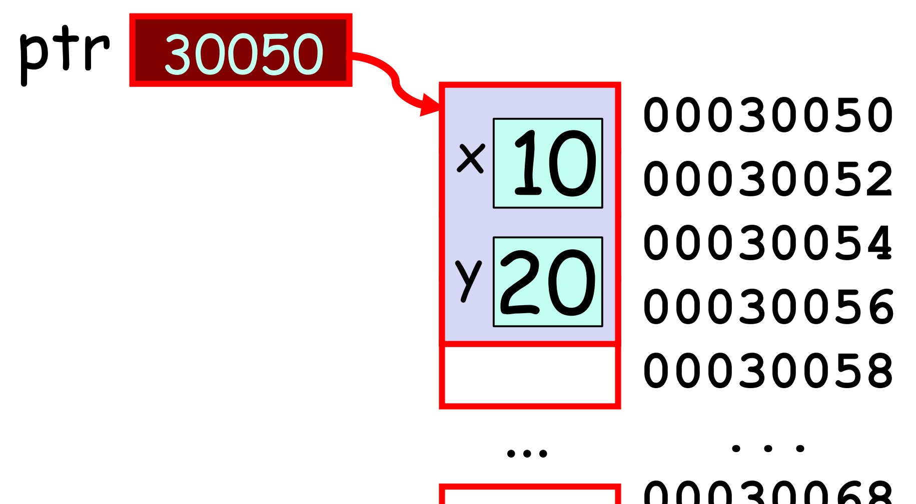
We can also use new and
create other types o

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

```
int main(void)
{
   // 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;
}
```

For instance, we can allocate an integer variable like this…

Or we can allocate a struct variable like this….



# New and Delete (

We can also use new and
create other types o

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

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

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

    // use our dyn    ic variable
    ptr->saySomet      erdy();

    // free ou
    delete ptr;

}
```

Then calls the Nerd constructor with these parameters to initialize it!

This allocates enough memory for a Nerd variable...

Or we can even allocate a class instance like this....

# Using new and delete in a class

```
class PiNerd
{
public:
    PiNerd(int n) {
      m_pi = new int[n]; // alloc array
      m_n = n;           // store its size!
      for (int j=0;j< m_n ;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;
};
```

So how we might use new/delete within a class?

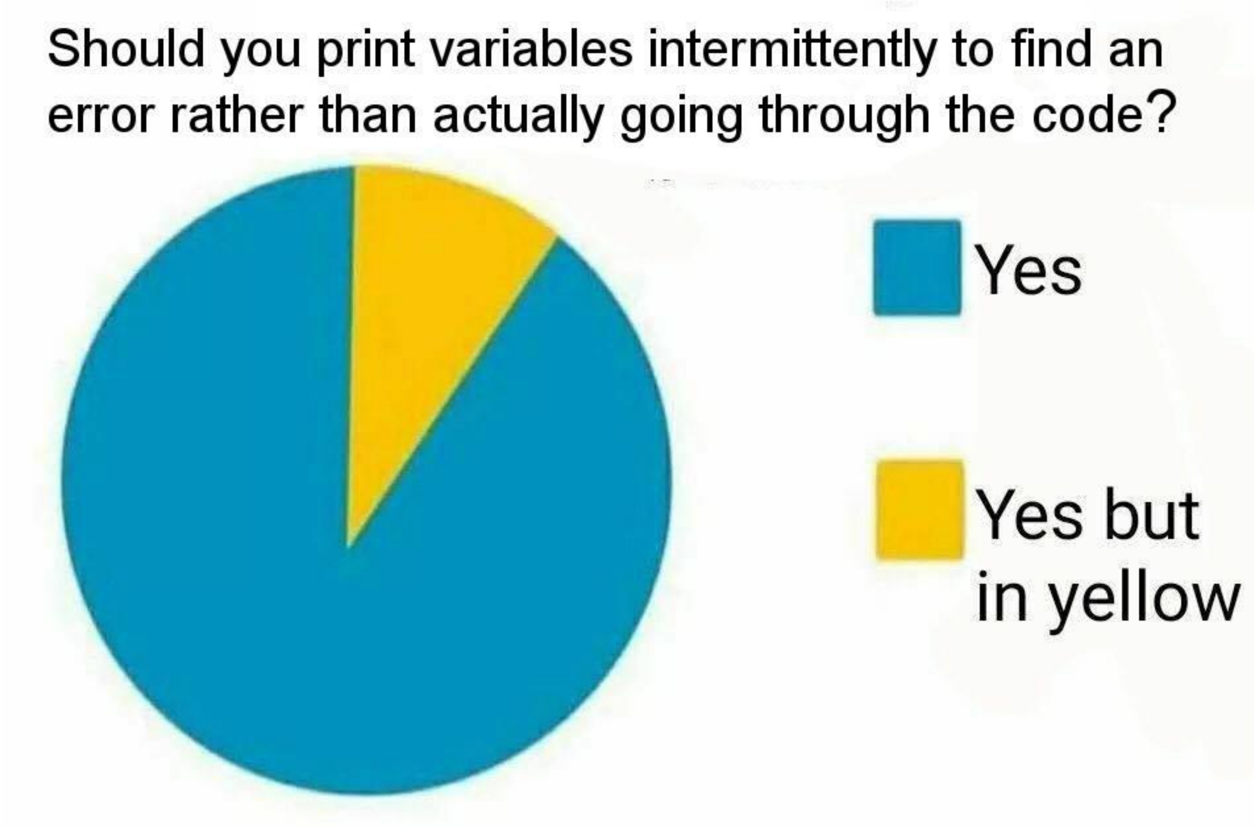
Well, here we have a class that represents people who like to memorize π – PiNerds!

As you can see, right now Pi Nerds can only memorize up to the first 10 digits of π.

Let's update our class so they can memorize as many digits as they like!

```
int main(void)
{
   PiNerd notSoNerdy(5);
   PiNerd superNerdy(100);

   notSoNerdy.showOff();
   superNerdy.showOff();
}
```

# Copy Construction




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

# Copy Construction…
# Why should you care?


Copy Construction is required in all nontrivial C++ programs.

If you fail to use it properly, it can result in nasty bugs and crashes.

So pay attention!

# Copy Construction

```
class Circ
{
public:
   Circ(int x, int y, int r)
   {
     m_x = x; m_y = y;  m_rad = r;
   }
  float GetArea(void) const;
private:
  float m_x, m_y, m_rad;
};
```

Last time we saw how to create a constructor function for a class...

Our simple constructor accepts three ints as arguments..

Question: Can constructors accept other types of variables as parameters?

Let's see...

# Copy Construction

```
class Point // an x,y coordinate
{
public:
    int m_x, m_
};

class Circ
{
public:
  Circ(const Point &pt, int rad )
  {
    m_x = pt.m_x;
    m_y = pt.m_y;
    m_rad = rad;
  }
  float GetArea(void) const;
private:
  float m_x, m_y, m_rad;
};
```

const means that our function can't modify the pt variable.

The & means "pass by reference" which is more efficient.

For example, what if I have a Point class like this...

If we like, we can define a Circ constructor that accepts a Point variable as an argument!

And of course, we still want our constructor to have a radius parameter...

Finally, we can write our constructor's body...

Allright, let's see it in action...

# Copy Construction

p

| m_x | 7 |
|---|---|
| m_y | 9 |

```
class Point // an x,y coordinate
{
public:
    int m_x, m_y;
};

class Circ
{
public:
  Circ(const Point &pt, int rad )
  {
    m_x = pt.m_x;
    m_y = pt.m_y;
    m_rad = rad;
  }
  float GetArea(void) const;
private:
  float m_x, m_y, m_rad;
};
```

c

```
class Circ
{
  ...                                  3
  Circ(const Point &pt,int rad)
  {
    m_x = pt.m_x;
    m_y = pt.m_y;
    m_rad = rad;
  }
  ...
private:
   m_x [   ] m_y [   ] m_rad [   ]
}
```

```
int main()
{
   Point p;
   p.m_x = 7;
   p.m_y = 9;

   Circle c(p,3);

   cout << c.getArea();
   ...
}
```

# Copy Construction

```
class Circ
{
public:
  Circ(int x, int y, int r)
  {
    m_x = x; m_y = y;  m_rad = r;
  }
  Circ(const Point &pt, int rad )
  {
    m_x = pt.m_x;
    m_y = pt.m_y;
    m_rad = rad;
  }
  Circ(const Circ &old )
  {
    m_x = old.m_x;
    m_y = old.m_y;
    m_rad = old.m_rad;
  }
  float GetArea(void) const;
private:
  float m_x, m_y, m_rad;
};
```

Ok, so we've seen a simple constructor…

And a constructor that accepts another class's variable…

What if we want to define a constructor for Circ that accepts another Circ variable??

This will allow us to initialize a new Circ variable (b) based on the value of an existing Circ variable (a).

Let's see how to do it!

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

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

# Copy Construction

This kind of thing is actually pretty useful… It lets us create a new variable with the same value as an existing variable.

Carey says: 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).

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(int x, int y, int rad)
 {
   m_x = x;
   m_y = y;
   m_rad = rad;
 }
 ...
private:
  m_x        m_y        m_rad
}
```

But wait! Circ variable b is accessing the private variables/functions of Circ variable a – isn't that violating C++ privacy rules?

```
  Circ(const Circ &old )
  {
    m_x = old.m_x;
    m_y = old.m_y;
    m_rad = old.m_rad;
  }
  float GetArea(void) const;
private:
  float m_x, m_y, m_rad;
};
```

```
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(int x, int y, int r)
  {
    m_x = x; m_y = y;  m_rad = r;
  }
  Circ(const Point &pt, int rad )
  {
    m_x = pt.m_x;
    m_y = pt.m_y;
    m_rad = rad;
  }
  Circ(const Circ &old )
  {
    m_x = old.m_x;
    m_y = old.m_y;
    m_rad = old.m_rad;
  }
  float GetArea(void) const;
private:
  float m_x, m_y, m_rad;
};
```

In C++ talk, this function is called a "copy constructor."

A copy constructor is a constructor function that is used to initialize a new variable from an existing variable of the same type.

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

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

# Copy Construction

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

```
int main()
{
  Circ a(1,1,5);

  Circ b(a);


  cout << b.GetArea();
}
```

A Copy Constructor is just like a regular constructor.

However, it takes another instance of the same class as a parameter instead of regular values.

# ...o... truction

This is a promise that you won't modify the oldVar while constructing your new variable!

This one's a bit more difficult to explain right now.

For now, just make sure you use an & here!

```
cl
{
pub
  Circ      oat x, float      float r)
  {
    m_x    x; m_y = y   m_rad = r;
  }
  Circ(const Circ & oldVar)
  {
    oldVar.m_x = 10; // error 'cause of const
    m_x = oldVar.m_x;
    m_y = oldVar.m_y;
    m_rad = oldVar.m_rad;
  }
  float GetArea(void)
  {
    return(3.14159*m_rad*m_rad);
  }
private:
  float m_x, m_y, m_rad;
};
```

The parameter to your copy constructor *should* be const!

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;
  }
  Circ( const Circ & oldVar)
  {
    m_x = oldVar.m_x;
    m_y = oldVar.m_y;
    m_rad = oldVar.m_rad;
  }
  float GetArea(void)
  {
    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 & oldVar)
  {
    m_x = oldVar.m_x;
    m_y = oldVar.m_y;
    m_rad = oldVar.m_rad;
  }
  float GetArea(void)
  {
    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 [ ]
}
```

We'r ... e
call
val

a

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

Now that our temp variable has been copy-constructed, it can be used normally by our foo function!

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

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

  foo(a);
}
```

Here's a simple program that passes a circle to a function...

Any guesses if/when the copy constructor is called?

# Copy Construction

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

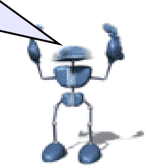
If you don't define your own copy constructor…

C++ will provide a default one for you…

It just copies all of the member variables from the old instance to the new instance…

This is called a "shallow copy."

But then why would I ever need to define my own copy constructor?



```
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;
};
```

When constructed, it uses new to dynamically allocate an array to hold the first N digits of π.

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.

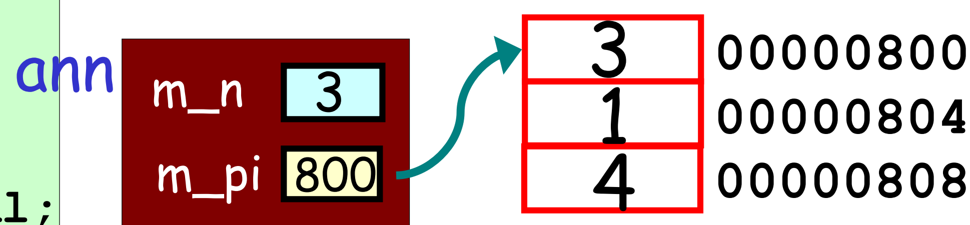
Let's see what happens when we use this class in a simple program.

And when it is destructed, it uses delete [] to release this array.

# Copy Construction

```
class PiNerd
{
public:           3
  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();
}
```



# ...struction

Now, watch what happens when we create our new ben variable and shallow copy ann's member variables into it…

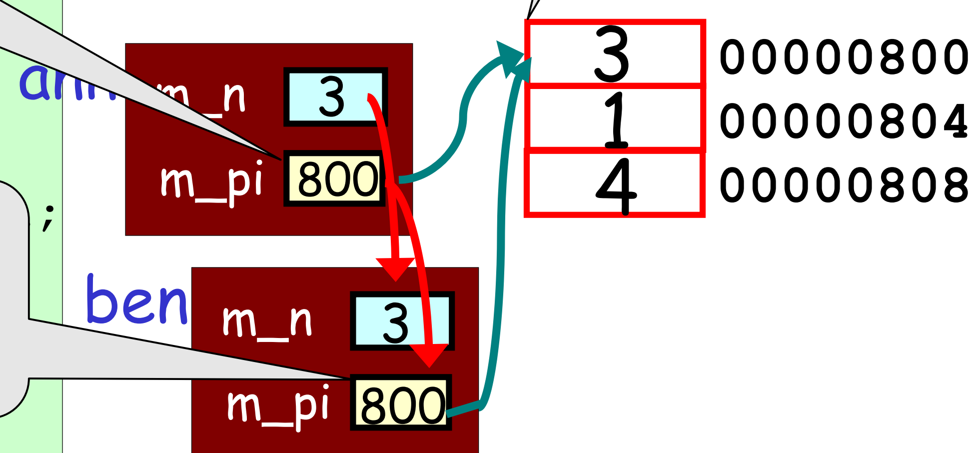
```
{
public:
   PiNerd(int n) {
     m_n = n;
     m_pi = new int[n];
     for (
       m_                                       ;
   }

   ~PiNerd()                                   }

   void ShowOff()
   {
     for (int j=0;j<m_n;j++)
       co                                       ;
    }
private:
    int *
};
```

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

   ann.ShowOff();
}
```

Point to ann's original copy of the array!

Both ann's m_pi pointer…

And ben's m_pi pointer…



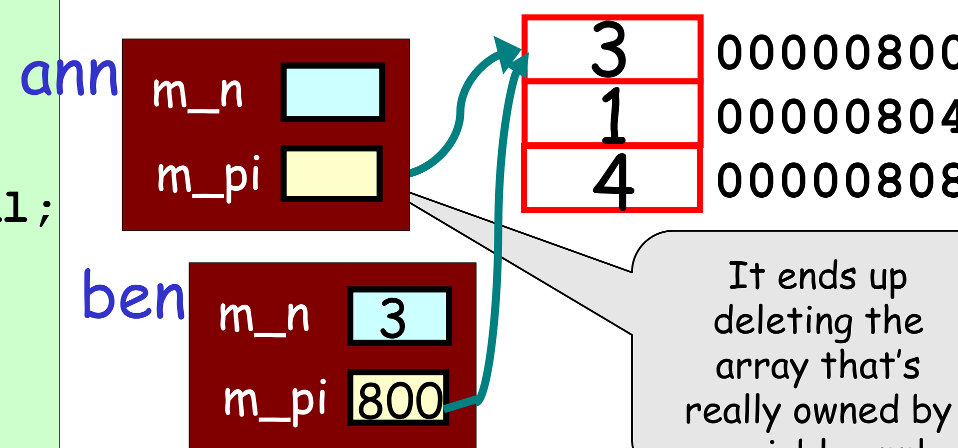
# struction

But that's a problem!

Because when ben is destructed…

```
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();
}
```




It ends up deleting the array that's really owned by variable ann!

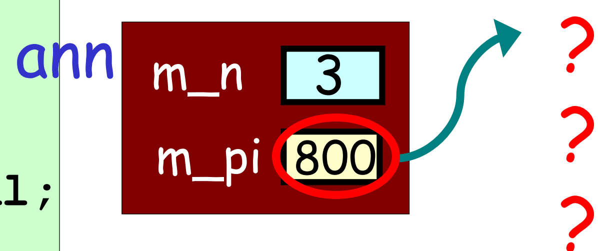
# Copy Construction

```
class PiNerd
{
public:
```

Because now, when we try to access ann's array, we get garbage!!!

```
   }

   ~PiNerd(){delete    _pi;}

   void ShowOff()
   {
     for (int j=0; <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

## …ral of the story?

```
class P…
{
public:
   PiNerd(int n) {
     m_n = n;
     m_pi = new int[n];
   …
   …i;}
   …
   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();
}
```

And you make a shallow copy of a class instance…

BAD THINGS will happen when you destruct either copy…

Any time your class holds pointer member variables*…

* or file objects (e.g., ifstream), network sockets, etc.

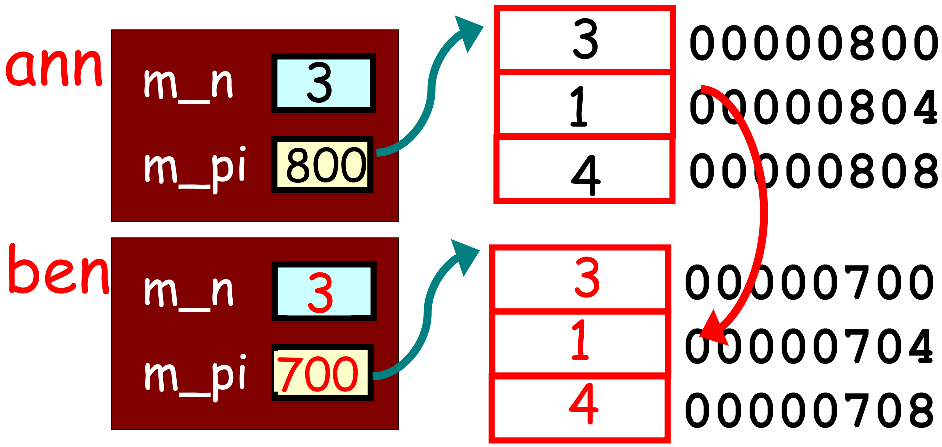
# Copy Construction

## 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

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



  }

  void showOff() { ... }

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

```
      ...
  }
```

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

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


  }

  void showOff() { ... }

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

```
   }

   ann.ShowOff();
}
```

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

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

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()

   ann.ShowOff();
}
```

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

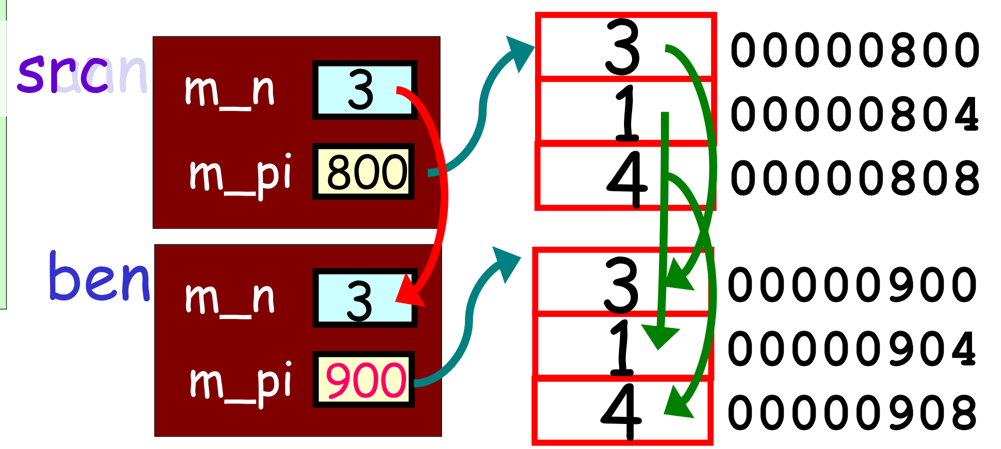
Finally, we have to manually copy over the contents of the original array to our new array.

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();
}
```



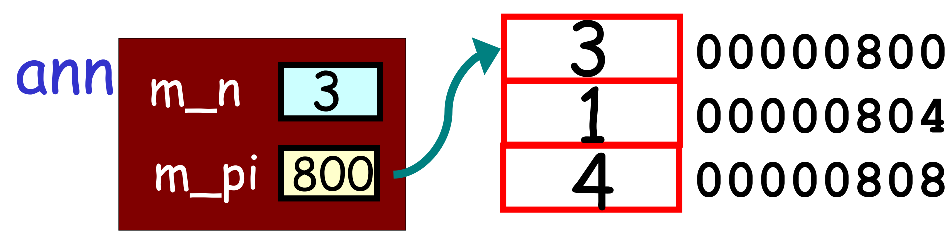

Let's watch our correct copy constructor work!

# 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;
     ...
  }// ben's d'tor called

  ann.ShowOff();
}
```

3
1
4




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