

Pointers

Topics

- Pointers
- Problem Solving and Testing Strategy

Pointers

- Pointers is not a so hard to understand
- Think of the concept of Pointers before thinking of how to code it
- Pointers enable very sophisticated operations
- First, we need to learn about l and r values.

Lvalues And Rvalues

- C Supports Two Kinds Of Expressions
- Lvalues (L for left side of assignment)
 - expressions which can be evaluated and modified
- Rvalues (R for right side of assignment)
 - expressions which can only be evaluated

Lvalue And Rvalue Examples

- Lvalue Examples:

- A Variable Name `int a;`

- An Array Index `array[0]`

- Rvalue Examples:

- Literal Constants `5.14e4`

- Arithmetic Expressions `5 * a`

Lvalue And Rvalue Examples

- Lvalue Examples:

- A Variable Name

```
int a;
```

- An Array Index

```
array[0]
```

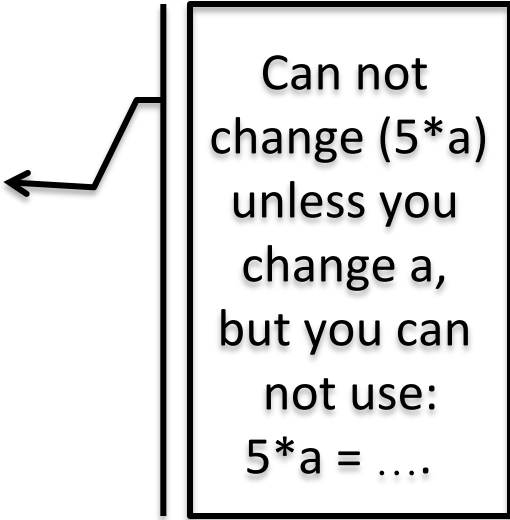
- Rvalue Examples:

- Literal Constants

```
5.14e4
```

- Arithmetic Expressions

```
5 * a
```



Can not
change (5*a)
unless you
change a,
but you can
not use:
5*a =

Lvalues

- An lvalue actually refers to a location in memory
 - We conveniently refer it by name

```
int a = 12;
```

Lvalues

- An lvalue actually refers to a location in memory
 - We conveniently refer it by name

```
int a = 12;
```

MEMORY
ADDRESS

1000

1004



...

Lvalues

- An lvalue actually refers to a location in memory
 - We conveniently refer it by name

```
int a = 12;
```

MEMORY
ADDRESS

1000

12

a

1004

...

Pointer Variables

- A pointer is a variable that contains the address of another variable

Pointer Variables

- A pointer is a variable that contains the address of another variable

```
int a = 12;  
int* intPtr;  
//  
//  
  
//  
//
```

Pointer Variables

- A pointer is a variable that contains the address of another variable

```
int a = 12;  
int* intPtr;  
// intPtr is a variable that will  
// contain a memory address of an int  
// marked by * after int  
  
//  
//
```

Pointer Variables

- A pointer is a variable that contains the address of another variable

```
int a = 12;  
int* intPtr;  
// intPtr is a variable that will  
// contain a memory address of an int  
// marked by * after int  
  
//  
//
```

Pointer Variables

- A pointer is a variable that contains the address of another variable

```
int a = 12;  
int* intPtr;  
// intPtr is a variable that will  
// contain a memory address of an int  
// marked by * after int  
intPtr = &a;  
// intPtr contains the address of var a
```

Pointer Variables

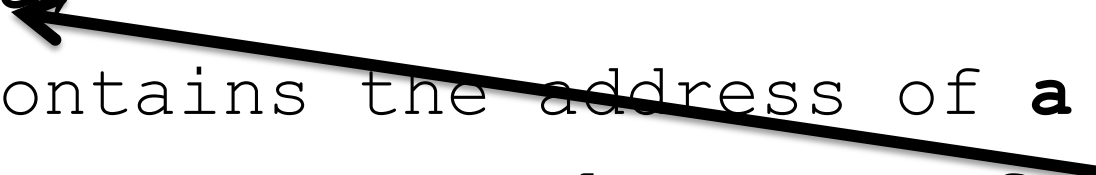
- A pointer is a variable that contains the address of another variable

```
int a = 12;  
int* intPtr;  
// intPtr is a variable that will  
// contain a memory address  
// marked by * after int  
intPtr = &a;  
//intPtr contains the address of var a  
//a's address not content because of &
```

Pointer Variables

- A pointer is a variable that contains the address of another variable

```
int a = 12;  
int* intPtr;  
// intPtr is a variable that will  
// contain a memory address  
// marked by * after int  
intPtr = &a;  
//intPtr contains the address of a  
// address not content because of &
```



Pointer Variables

- A pointer is a variable that contains the address of another variable

```
int a = 12;  
// means a in RAM should contain 12  
//
```

Pointer Variables

- A pointer is a variable that contains the address of another variable

```
int a = 12;
```

```
//
```

```
//
```

MEMORY
ADDRESS

1000

1004



a

...

Pointer Variables

- A pointer is a variable that contains the address of another variable

```
int a = 12;
```

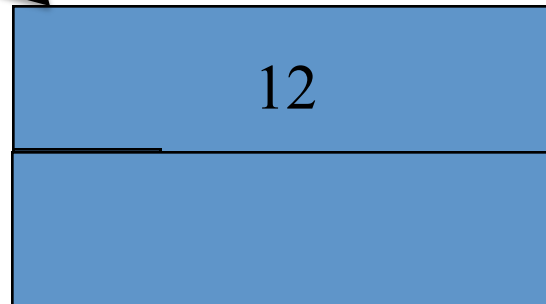
```
//
```

```
//
```

MEMORY
ADDRESS

1000

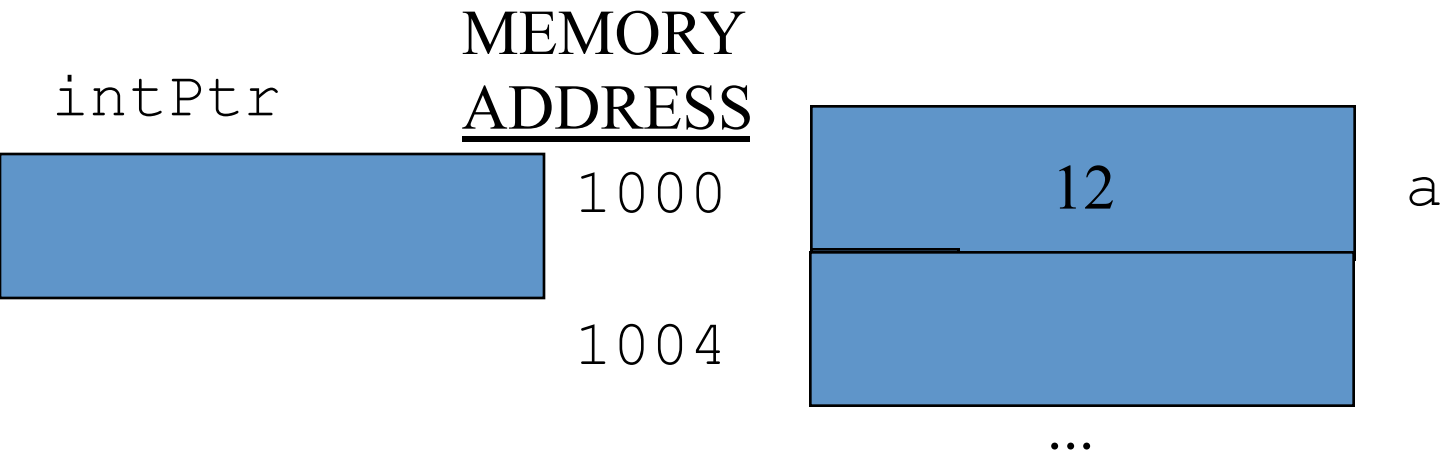
1004



Pointer Variables

- A pointer is a variable that contains the address of another variable

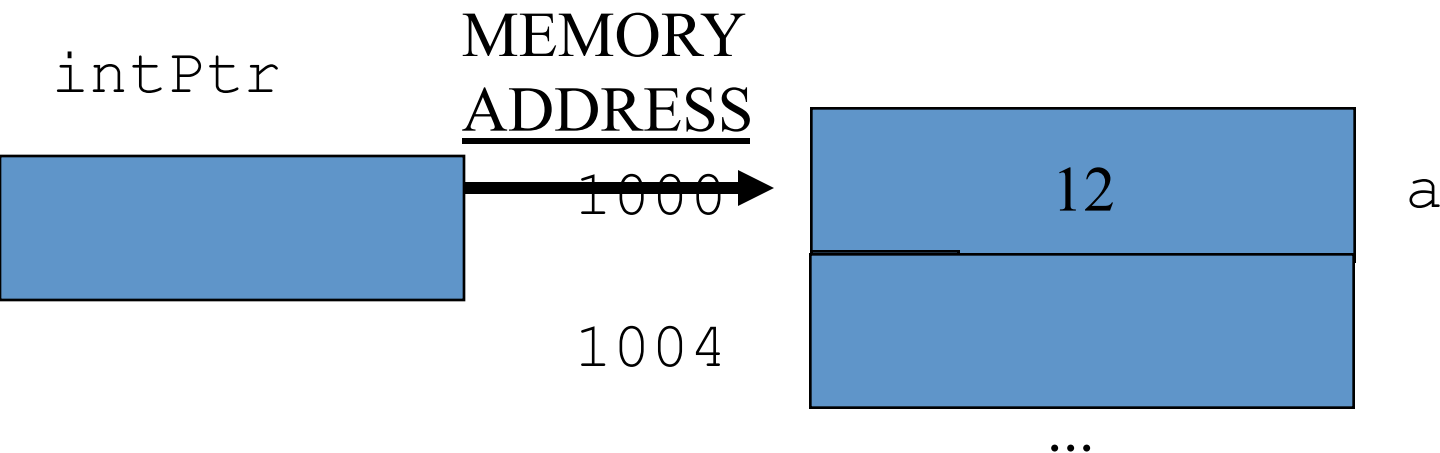
```
int a = 12;  
int* intPtr;  
intPtr = &a;
```



Pointer Variables

- A pointer is a variable that contains the address of another variable

```
int a = 12;  
int* intPtr;  
intPtr = &a;
```



Pointer Variables

- Like any kind of variable, pointers must be declared: *typename* varname;*
- Once declared, you can change the address the pointer contains by assigning it using the address of operator &
- To change the content of the address the pointer contains you use the *

Pointer Variables

- Declare: `int* intP;`
- Assign the address of another variable to intP:
`intP = &x; //x must be same type`
- To change the content of x:
`*intP = 5 //x now contains 5`

Pointer Variables

- Like Any Kind Of Variable, Pointers Must Be Declared: *typename* varName;*

```
double d = 13.1;
```

```
double* dPtr; // to declare
```

```
dPtr = &d;    // to assign
```


Pointer Variables

- Like Any Kind Of Variable, Pointers Must Be Declared: *typename* varName;*

```
double d = 13.1;
```

```
double* dPtr;
```

```
dPtr = &d;
```

MEMORY
ADDRESS

2000

2008



...

Pointer Variables

- Like Any Kind Of Variable, Pointers Must Be Declared: *typename* varName;*

```
double d = 13.1;
```

```
double* dPtr;
```

```
dPtr = &d;
```

MEMORY
ADDRESS

2000

2008



d

...

Pointer Variables

- Like Any Kind Of Variable, Pointers Must Be Declared: *typename* varName;*

```
double d = 13.1;
```

```
double* dPtr;
```

```
dPtr = &d;
```

MEMORY
ADDRESS

2000

13.1

d

2008

...

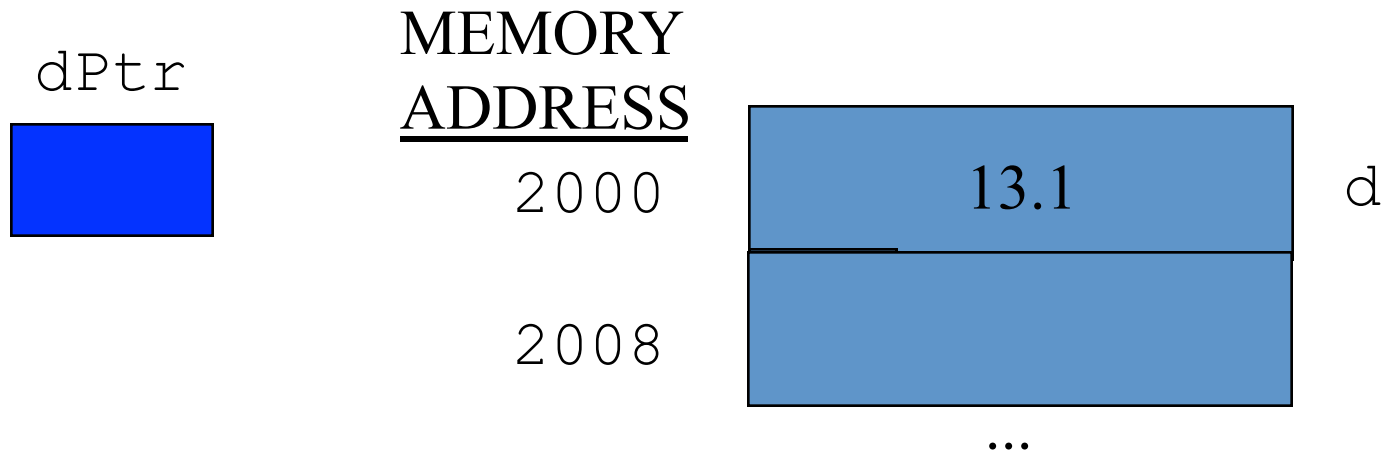
Pointer Variables

- Like Any Kind Of Variable, Pointers Must Be Declared: *typename* varName;*

```
double d = 13.1;
```

```
double* dPtr;
```

```
dPtr = &d;
```



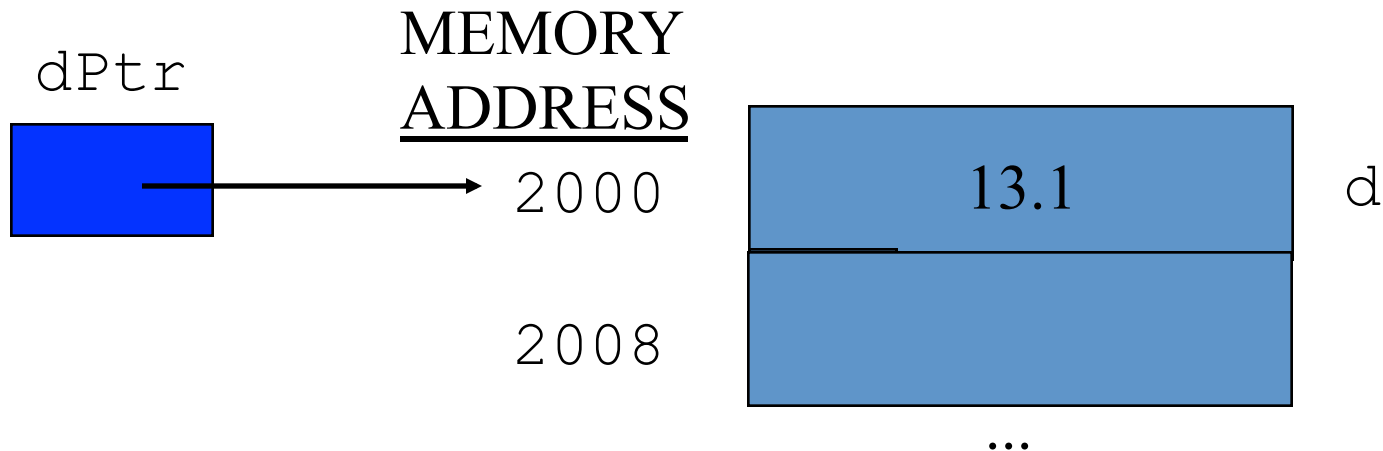
Pointer Variables

- Like Any Kind Of Variable, Pointers Must Be Declared: *typename* varName;*

```
double d = 13.1;
```

```
double* dPtr;
```

```
dPtr = &d;
```



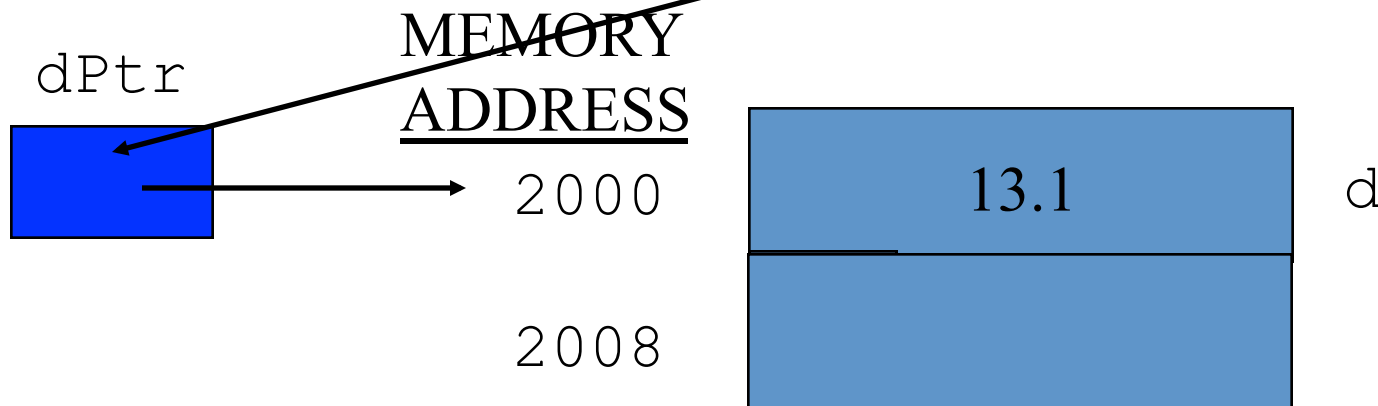
Pointer Variables

- Like Any Kind Of Variable, Pointers Must Be Declared: *typename* varName;*

```
double d = 13.1;
```

```
double* dPtr;
```

```
dPtr = &d;
```



The Pointer Is
An Address We
Can Change

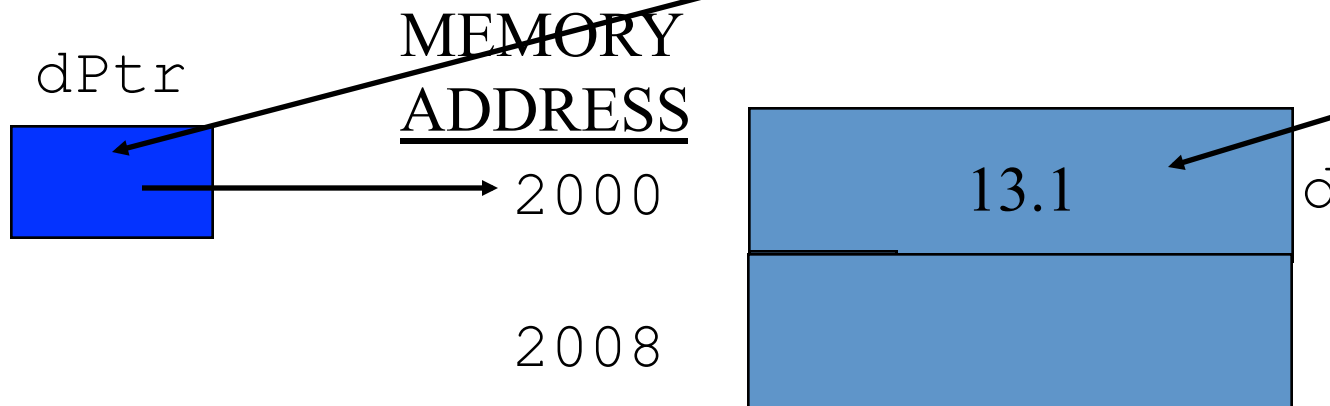
Pointer Variables

- Like Any Kind Of Variable, Pointers Must Be Declared: *typename* varName;*

```
double d = 13.1;
```

```
double* dPtr;
```

```
dPtr = &d;
```



The Pointer Is
An Address We
Can Change

We Can Change
What The Pointer
Points To

Pointer Variables

- Once declared, a pointer points to variables of the same pointer's declared type

For example:

```
double d = 13.1; // d is double
```

```
double* dPtr; // dPtr to a double
```

- After all the pointer “points to” a RAM address that contains a value, which must be of some type

Pointer Variables

- Once declared, a pointer points to variables of the same pointer's declared
- For example:

```
double d = 13.1; // d is double
```

```
double* dPtr; // dPtr to a double
```

```
dPtr = &d; // dPtr points to d
```

Pointer Variables

- The variable the pointer points to is called its ***referent***
 - *In the previous example, d is the referent*
- The content of the pointer is another RAM address
- That RAM address may contain a value that can be changed just like we did before, using assignment statements

Pointer Dereferencing

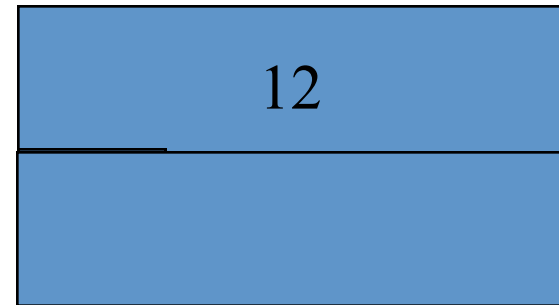
- The Thing A Pointer Points To Can Be Manipulated By The Pointer Variable

```
int a = 12;  
int* intPtr;  
intPtr = &a;  
*intPtr = 5;
```

MEMORY
ADDRESS

1000

1004



a

...

Pointer Dereferencing

- The Thing A Pointer Points To Can Be Manipulated By The Pointer Variable

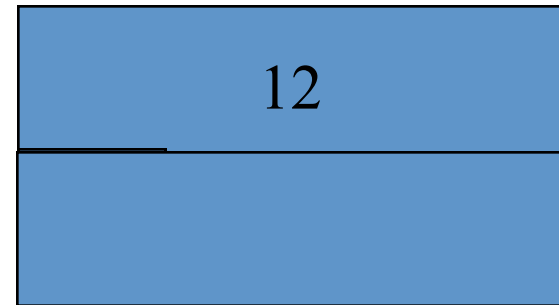
```
int a = 12;  
int* intPtr;  
intPtr = &a;  
*intPtr = 5;
```

MEMORY
ADDRESS

1000

1004

intPtr



a

Pointer Dereferencing

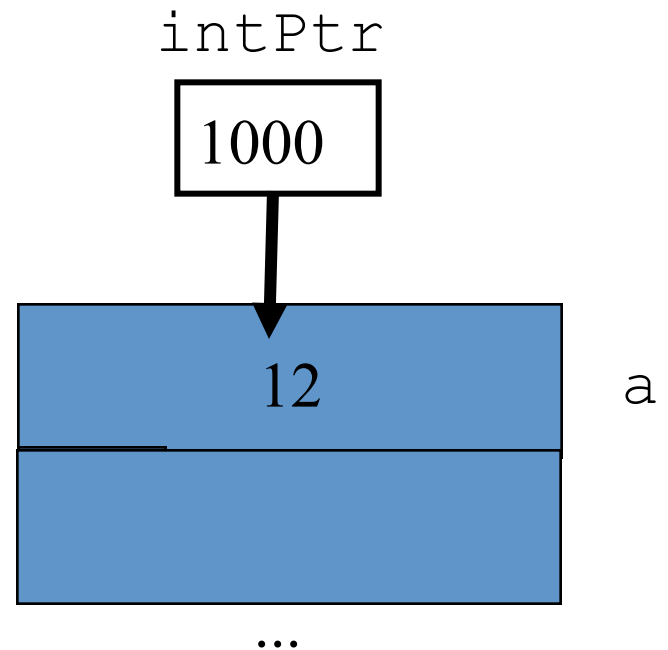
- The Thing A Pointer Points To Can Be Manipulated By The Pointer Variable

```
int a = 12;  
int* intPtr;  
intPtr = &a;  
*intPtr = 5;
```

MEMORY
ADDRESS

1000

1004



Pointer Dereferencing

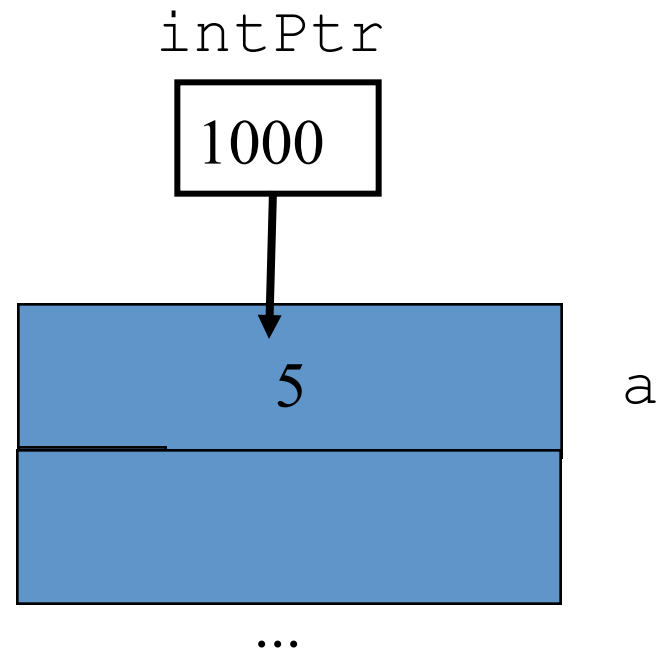
- The Thing A Pointer Points To Can Be Manipulated By The Pointer Variable

```
int a = 12;  
int* intPtr;  
intPtr = &a;  
*intPtr = 5;
```

MEMORY
ADDRESS

1000

1004



Pointer Dereferencing or Indirection

- The change the RAM content the pointer is pointing to, use an assignment statement, but add *
- * goes before the pointer variable.
- * is the dereference or indirection operator
 - It traverses the pointer to access what is being pointed to

Pointer Dereferencing

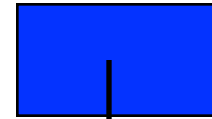
```
int a = 12;  
int* intPtr;  
intPtr = &a;  
*intPtr = 5;
```

MEMORY
ADDRESS

1000

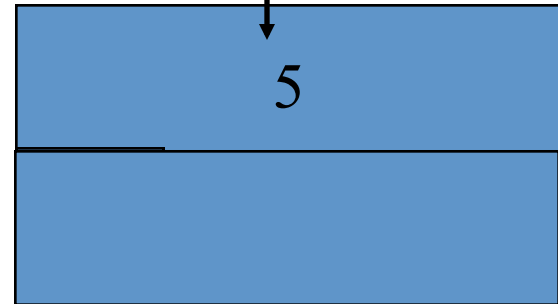
1004

intPtr



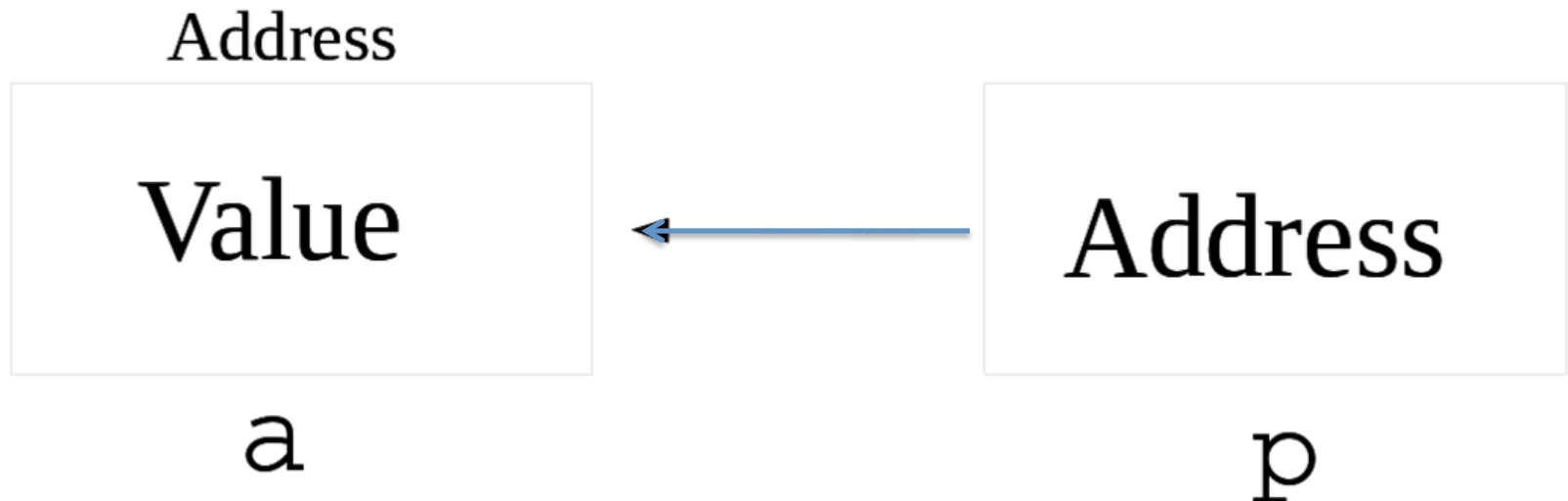
5

a



Stop Here – Read the Book

- Read the section on Pointers and Arrays on declaring and assigning pointers - ignore structures
- Carefully read Pointer Dereferencing then



Now Practice

- The code:

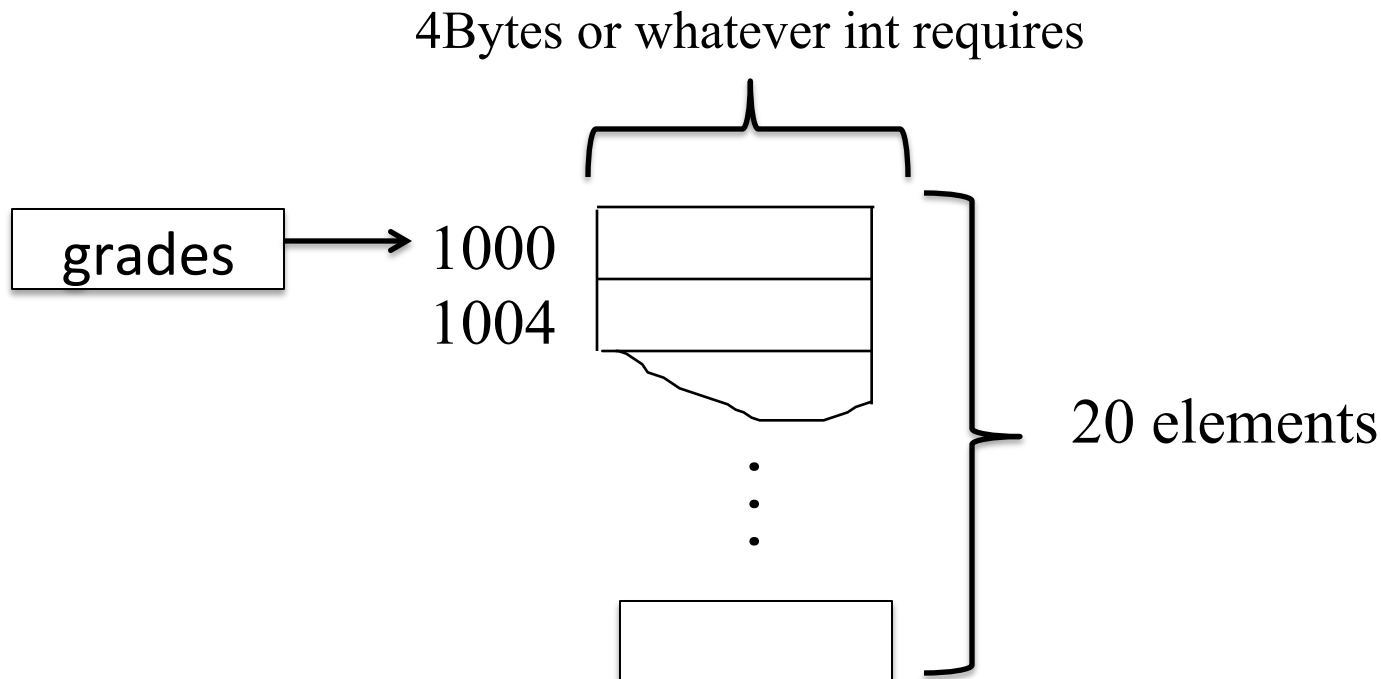
```
int x = 12;  
    //use printf to print content of x  
int* intPtr;  
intPtr = &x;  
*intPtr = 5;  
    //use printf to print content of x
```

Pointers and Arrays

- It sounds funny to differentiate a pointer from an array
- An array is a pointer!

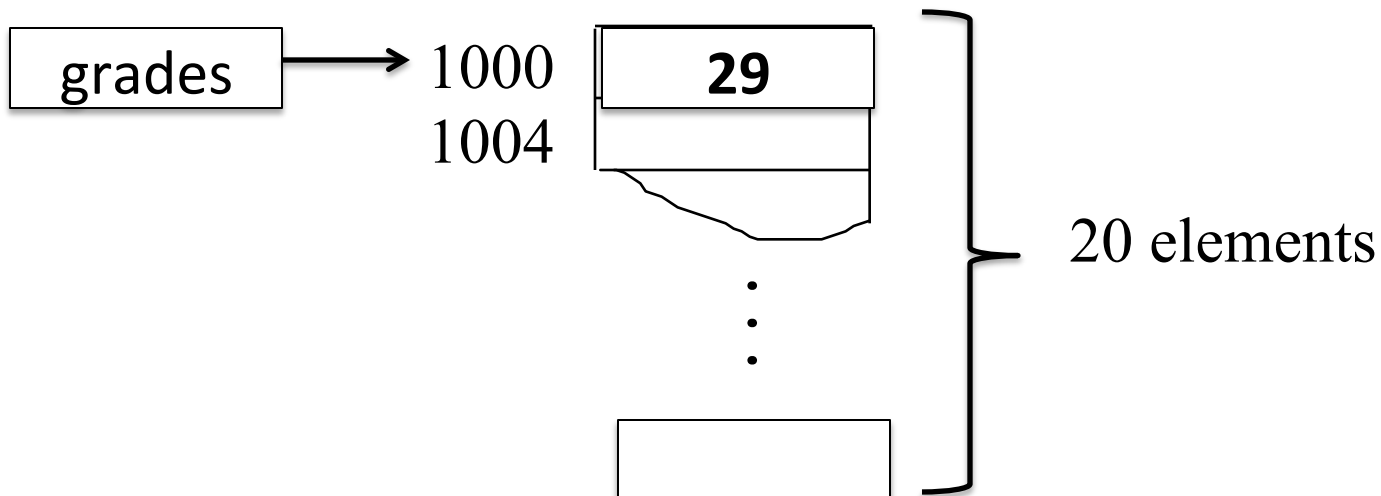
Pointers and Arrays

`int grades[20];` //grades is a pointer to some RAM location



Pointers and Arrays

```
int grades[20];  
grades[0]=29;
```



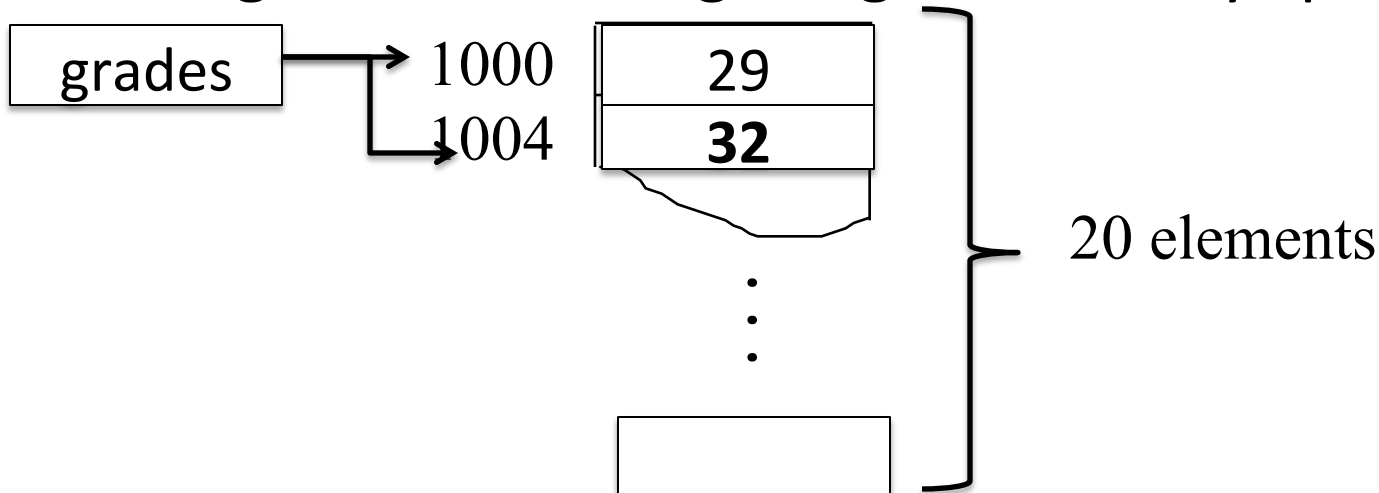
Pointers and Arrays

```
int grades[20];
```

```
grades[0]=29;
```

```
grades[1]=32; //note the arrow moved to next location
```

grades[n] works by starting from address 1000, then adding n locations. Again, grades always points to [0].



Pointers and Arrays

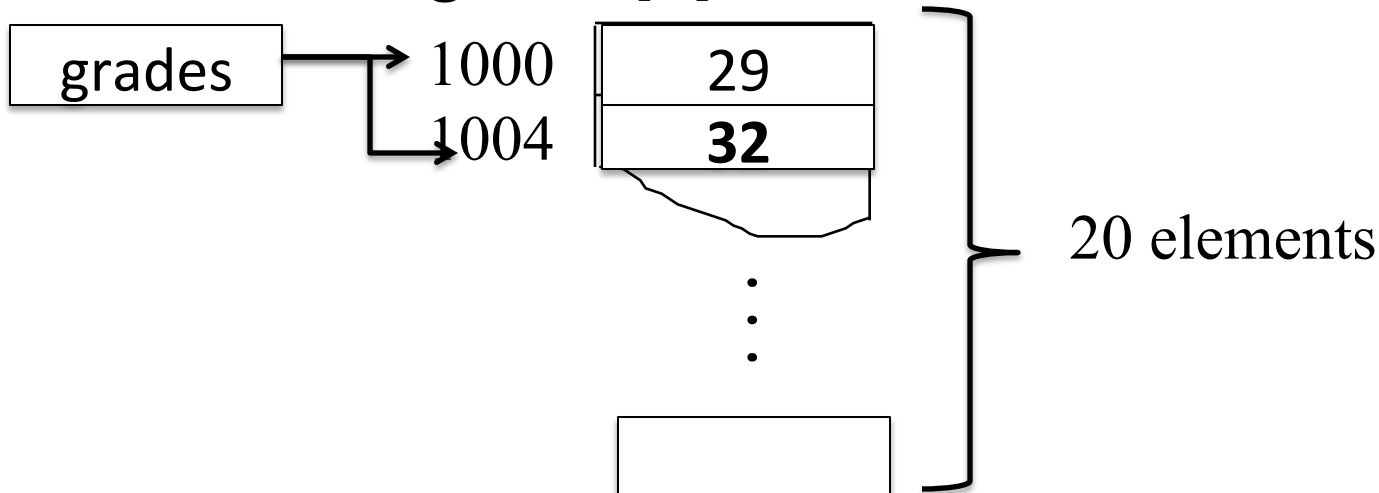
```
int grades[20];
```

```
grades[0]=29;
```

```
grades[1]=32; //note the arrow moved to next location
```

Another way to view grades[n] is: $*(grades+n)$

But we find grades[n] to be easier to use

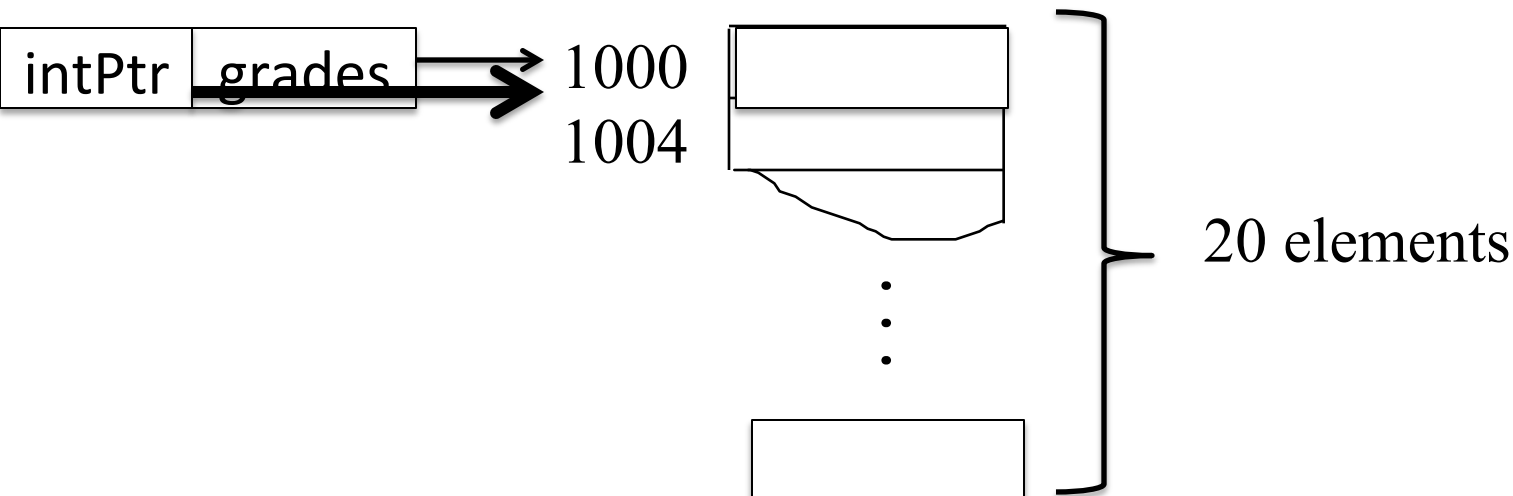


Pointers and Arrays

```
int grades[20];
```

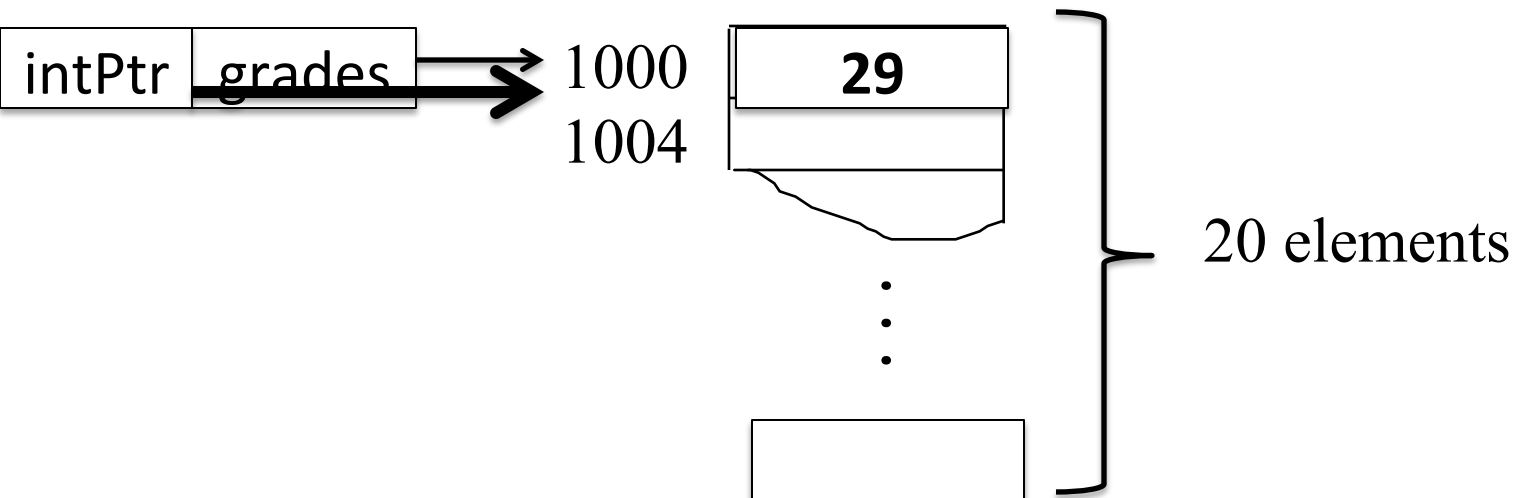
```
int* intPtr; //declare a pointer to an integer
```

```
intPtr = &grades; //makes intPtr point to grades[0]
```



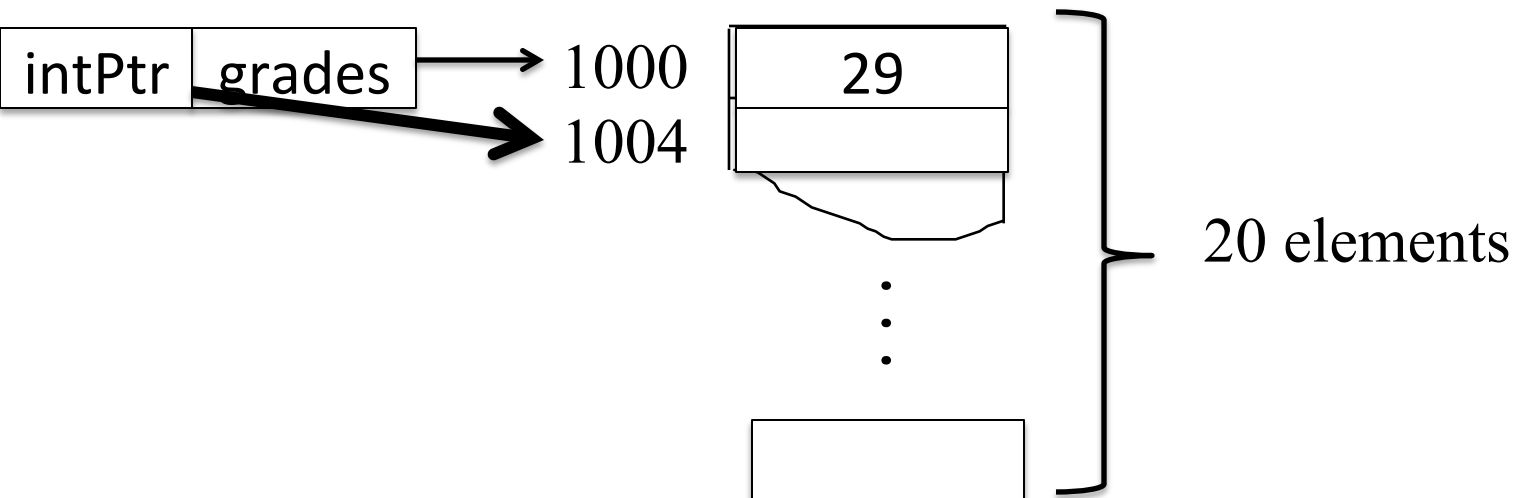
Pointers and Arrays

```
int grades[20];  
int* intPtr;  
intPtr = &grades;  
*intPtr = 29;
```



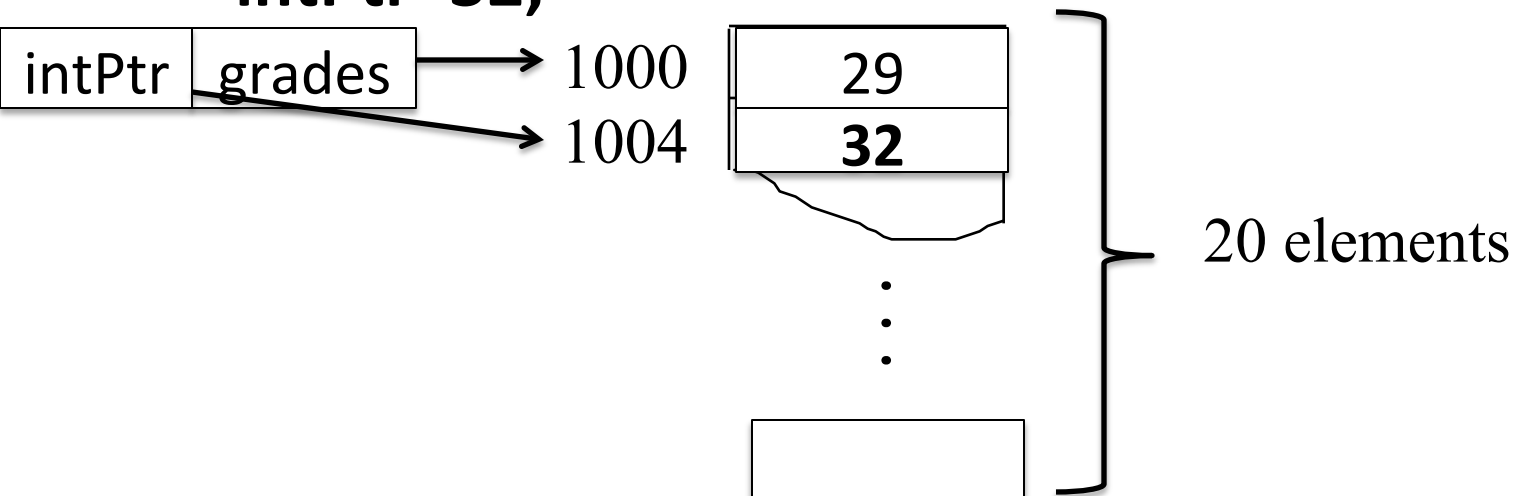
Pointers and Arrays

```
int grades[20];  
int* intPtr;  
intPtr = &grades;  
*intPtr = 29;  
intPtr++;
```



Pointers and Arrays

```
int grades[20];  
int* intPtr;  
intPtr = &grades;  
*intPtr = 29;  
intPtr++;  
*intPtr=32;
```



Pointers and Arrays

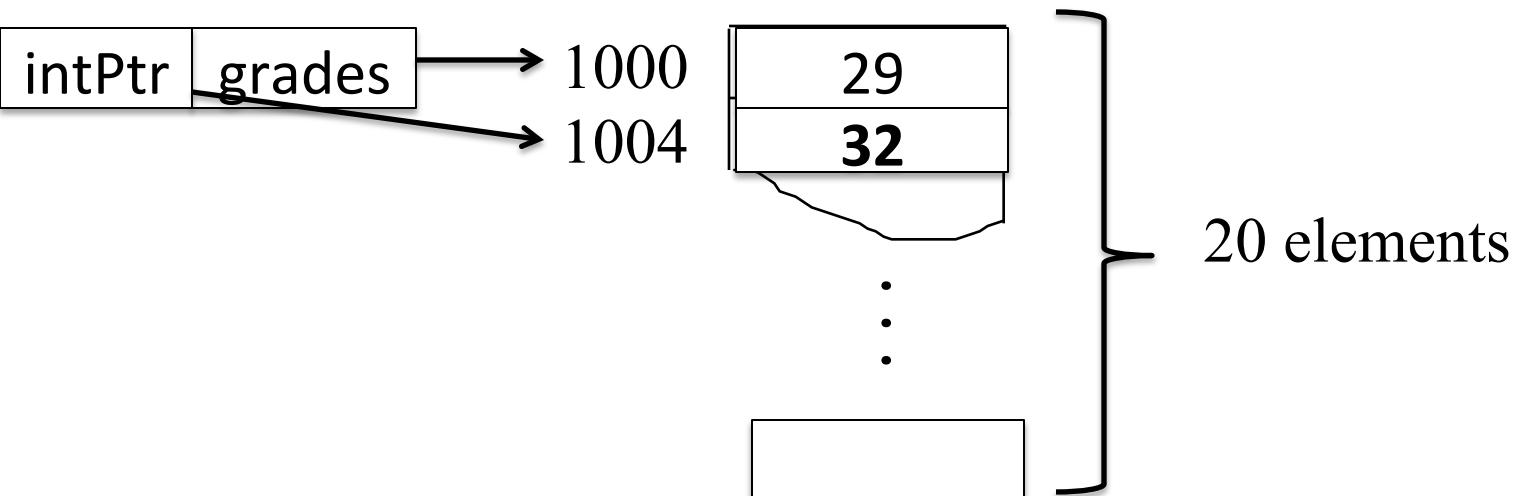
```
int grades[20];
```

```
int* intPtr;
```

```
intPtr = &grades;
```

```
*intPtr = 29;
```

```
*(intPtr+1)=32; //You can also do this instead
```



So far

- Declare the pointer as a type
- The pointer can only point to other variables of its same type
- You can assign using the address of
- You can change the content the pointer points to using *

Additionally

- Declare the pointer as a type
- The pointer can only point to other variables of its same type
- You can assign using the address of
- You can change the content the pointer points to using *
- **To change the content of the pointer (where it is point to), use regular assignment and math (e.g. pointing to arrays)**

More on Pointers - FYI

- Dynamic Arrays – filled with fun and danger!
- Covered in CS 52 (C++)
- Mostly done to manage RAM – hence most O.S. are written in C or C++
- involves malloc, free, and sizeof functions

Parameter Passing

- So far, our functions cannot alter their parameters
 - Referred to as “pass-by-value”
 - These functions can only provide a single output value
- However, there is another kind of parameter passing scheme that involves pointer variables
 - Referred to as “pass-by-reference”

Parameter Passing

- Passing by reference means that the argument address in RAM is passed into the parameter.
- Recall that passing by value, which we used all along, means that the argument content (literal) is copied into the parameter.

Parameter Passing

- Reference parameters are not copies of the argument, but “point to the same thing” the argument contains in RAM.
- Hence, a reference parameter is the argument itself
- Parameter must be a variable
- Specified when the prototype uses the syntax:
`type *`
- Caller needs to send the data by saying:
`& variable`
- Code will clarify in the next slides

Parameter Passing

- Anytime you want a function to change the caller's variable, you need to use pointers and pass-by-reference in C
- We've already seen this with `scanf (...)`
- After you type input, `scanf` assigns the RAM location of the `&variable` to the input

Parameter Passing

- Anytime you want a function to change the caller's variable, you need to use pointers and pass-by-reference in C
- We've already seen this with `scanf (...)`

```
int i;  
scanf( &i );
```

Parameter Passing

- We had to say `&i` because we want `scanf (...)` to change our value of `i`.
 - This requires pass-by-reference
- The `&i` creates a pointer to an `int`
- Officially, `&` is the “address operator” and gets its memory location
 - A pointer variable holds the address of another variable (an l-value)

Parameter Passing

- Reference Parameter Example:

```
void swap(int * x, int * y) {  
    int temp = *x;  
    //temp contains whatever x points to  
    *x = *y; //x and y point to same thing  
    *y = temp; //so x will change?  
}
```

- Legal Invocation???

```
int i=0, j=20;  
swap( &i, &j );  
//print i and j here, did they change?
```

Parameter Passing

- Reference Parameter Example:

```
void swap(int * x, int * y) {  
    int temp = *x;  
    *x = *y;  
    *y = temp;  
}
```

- Legal Invocation??? Why?

```
int i=0, j=20;  
swap( i, j++ );
```

Parameter Passing

- Reference Parameter Example:

```
void swap(int * x, int * y) {  
    int temp = *x;  
    *x = *y;  
    *y = temp;  
}
```

- Legal Invocation??? Why?

```
int i=0, j=20;  
swap( &7-10, &i/j );
```


Function Call And Return

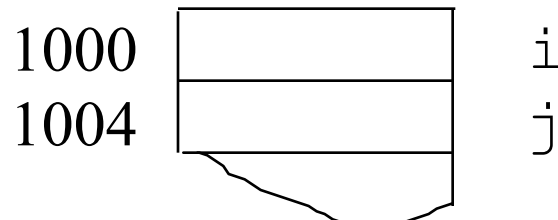
```
void swap( int *x, int *y);  
main( )
```

```
int i = 0, j = 20;  
  
swap( &i, &j );  
  
return 0;
```

```
void swap( int* x,  
          int* y)
```

```
int *temp = *x;  
*x = *y;  
*y = *temp;
```

Memory Model



Function Call And Return

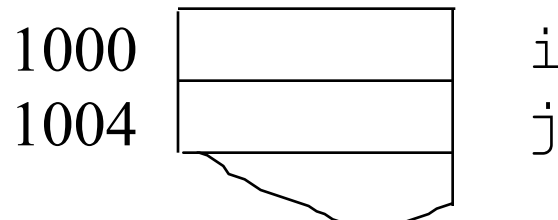
```
void swap( int *x, int *y);  
→ main( )
```

```
int i = 0, j = 20;  
  
swap( &i, &j );  
  
return 0;
```

```
void swap( int* x,  
          int* y)
```

```
int *temp = *x;  
*x = *y;  
*y = *temp;
```

Memory Model



Function Call And Return

```
void swap( int *x, int *y);  
→ main( )
```

```
int i = 0, j = 20;  
↓  
swap( &i, &j );  
  
return 0;
```

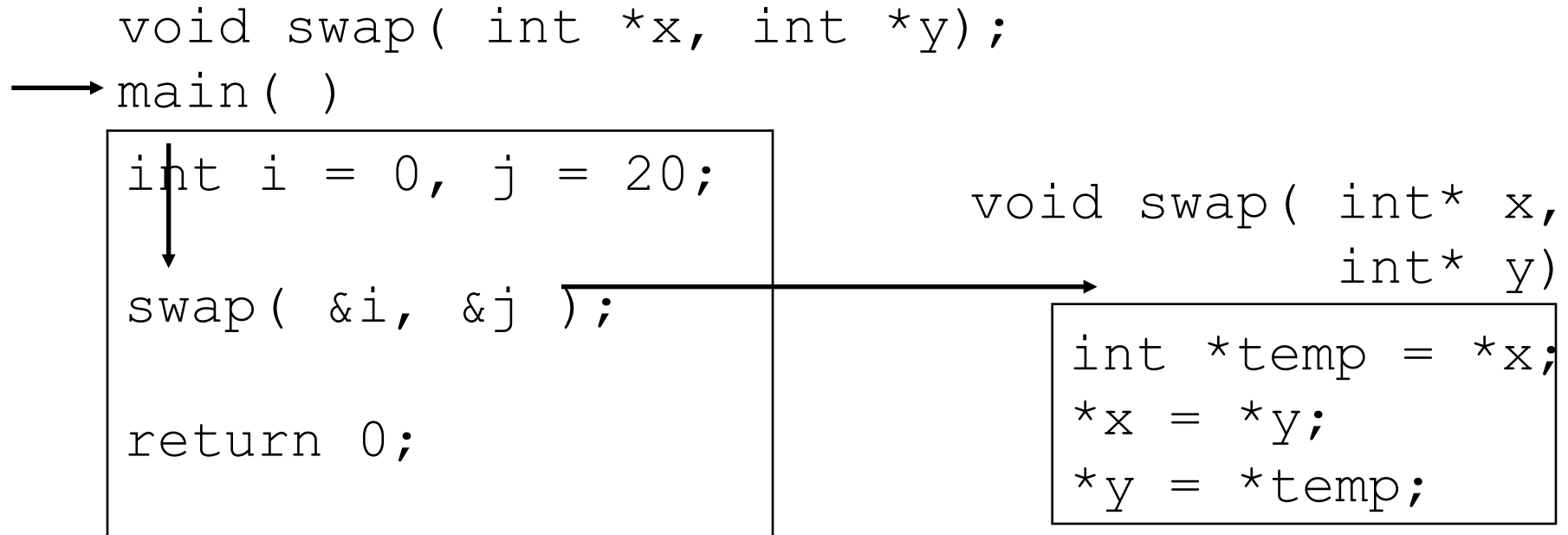
```
void swap( int* x,  
          int* y)
```

```
int *temp = *x;  
*x = *y;  
*y = *temp;
```

Memory Model

1000	0	i
1004	20	j

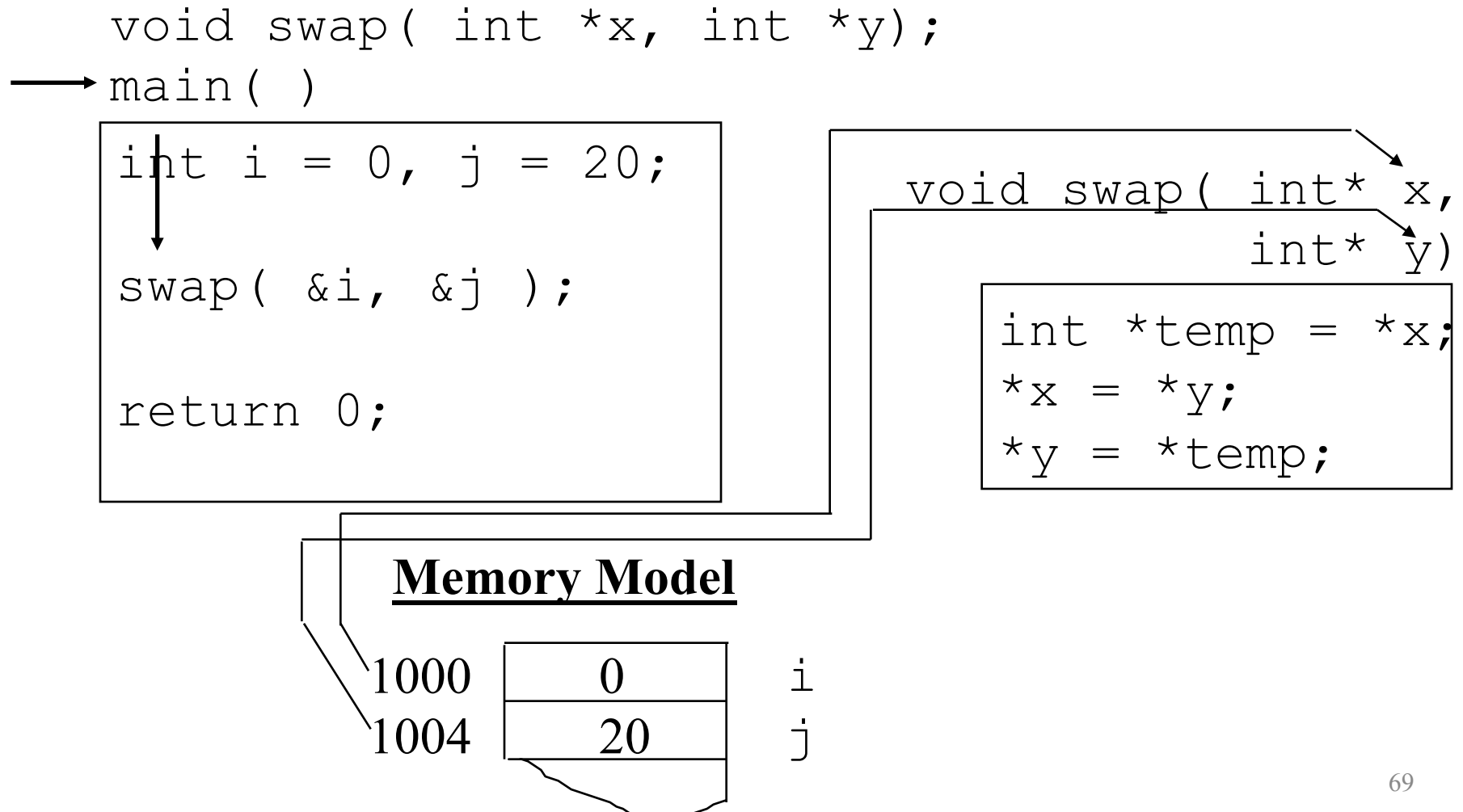
Function Call And Return



Memory Model

1000	0	i
1004	20	j

Function Call And Return



Function Call And Return

```
void swap( int *x, int *y);  
→ main( )
```

```
int i = 0, j = 20;  
↓  
swap( &i, &j );  
  
return 0;
```

```
void swap( int* x,  
           int* y)
```

```
int *temp = *x;  
*x = *y;  
↓  
*y = *temp;
```

Memory Model

1000

20

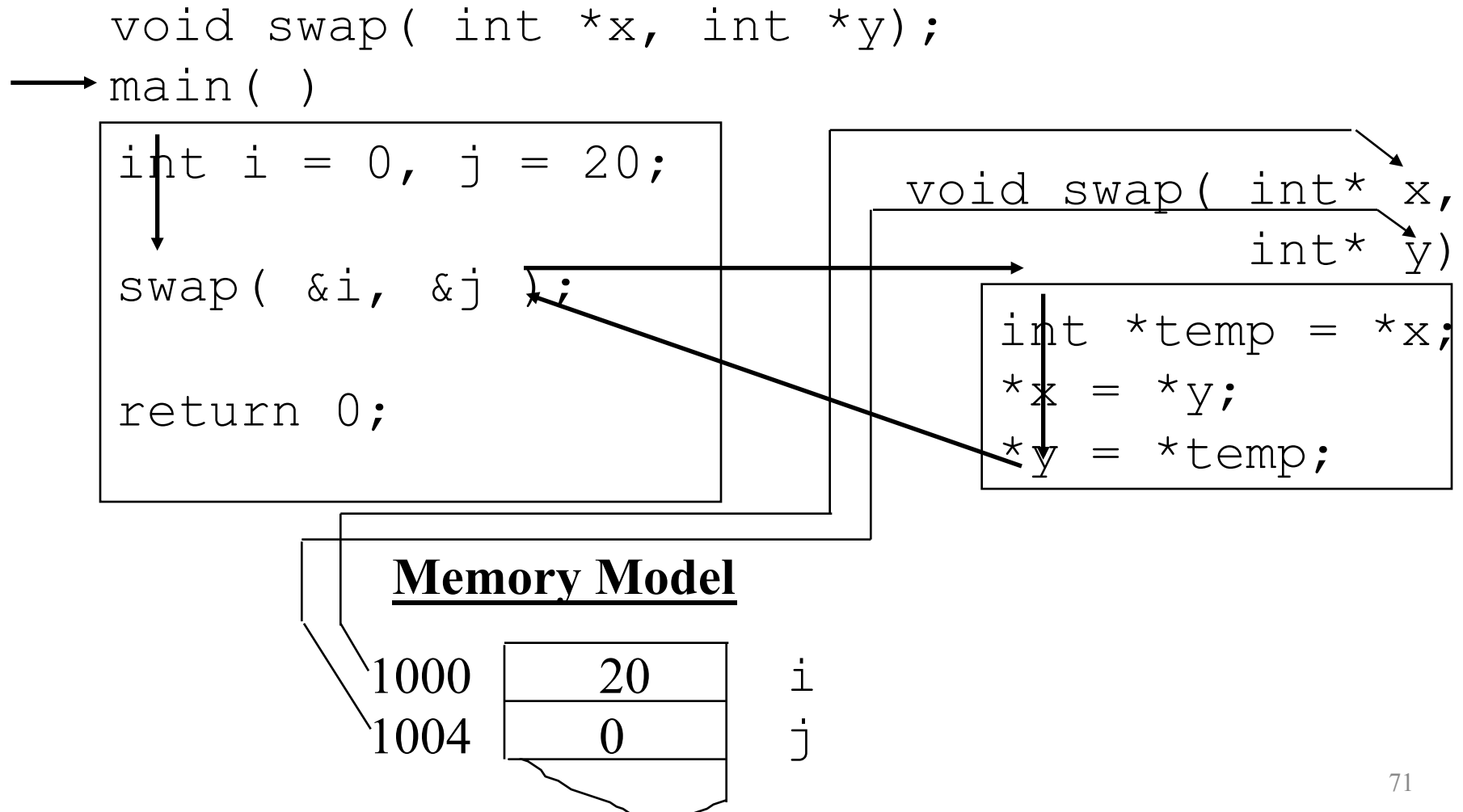
i

1004

0

j

Function Call And Return



Function Call And Return

```
void swap( int *x, int *y);  
→ main( )
```

```
int i = 0, j = 20;  
↓  
swap( &i, &j );  
↓  
return 0;
```

```
void swap( int* x,  
          int* y)
```

```
int *temp = *x;  
*x = *y;  
*y = *temp;
```

Memory Model

1000	20	i
1004	0	j

Function Call And Return

```
void swap( int *x, int *y);  
→ main( )
```

```
int i = 0, j = 20;  
↓  
swap( &i, &j );  
↓  
return 0;  
←
```

```
void swap( int* x,  
          int* y)
```

```
int *temp = *x;  
*x = *y;  
*y = *temp;
```

Memory Model

1000	20	i
1004	0	j

Summarizing Parameter Passing

- The caller passes the address of actual reference parameters to invoked functions

Summary of Operations with Pointers

- Point to another variable using address of: **&**
- Change the content of other variables using: *****
- In place of an **array** – or as an array: pointer variable points to array name – then instead of using an array index, advance the pointer
- In functions where more than one value is to be returned. Pass arguments **by reference**.

Swap Demo

```
#include <stdio.h>
```

/* In C, you pass parameters to a function using a pass-by-reference scheme by using pointers which are declared using *. Reference parameters are ones such that if the function changes it value, the caller will see those changes. As a result, the "real" value is passed to the function, rather than a copy, as would occur when passing by value as we learned to do originally.

Reference parameters are used when a function wants to change a value and wants to be sure the driver code "sees" that changed value.

```
*/
```

```
void swapper( int * i, int * j );
```

```
int main( ) {  
    int i, j;
```

```
    printf( "Please supply two ints to swap...\n" );  
    scanf( "%d %d", &i, &j );  
    printf( "Before swapping, i = %d and j = %d\n", i, j );  
    /*
```

You send reference parameters similar to the way we use scanf, by prefacing the variables with the &. This address operator converts a variable into its address or location.

```
*/  
    swapper( &i, &j );  
    printf( "After swapping, i = %d and j = %d\n", i, j );  
    return( 0 );  
}
```

```
void swapper( int * i, int * j ) {  
    int temp = * i;  
    *i = *j;  
    *j = temp;  
}
```

Summarizing Swap

- Pass-by-value results in copies being made of every argument
 - This might have a performance impact on your code
- However, pass-by-reference makes things more complex
 - Your function may have unintended side effects, since it can change values inside the caller's world

Mixing Parameter Types

- A function may use both kinds of parameter passing schemes in one prototype

```
void process ( int input, int* output );
```



this parameter
passed by value



this parameter
passed by reference

Done This Before

- Reminder of Problem Solving and Testing Strategy

Problem Solving Strategy

- One big problem is harder to solve than many smaller problems
- Understand the problem
 - what result is expected
 - what process can provide these results
 - what parameters are needed for these processes
 - write function descriptions in english telling what the function should do

Problem Solving Strategy

- C Syntax Typically Obscures Understanding
 - write out your solution on paper FIRST
 - use flow charts or pseudocode
 - translate to C syntax on paper
 - try not to compose code at a terminal
- Great Answers Don't Come The First Time
 - iteratively refine and enhance partial solutions

Testing Strategy

- How Do You Test Functions?
 - Test One Function At A Time
 - Display Intermediate Results
 - You May Need To Create Test Data To Use Via “Driver Programs”
 - If The Function Being Tested Calls Other Functions, Create “Stubs”
 - Try Varying One Thing At A Time
 - if something goes wrong, you know what changed

Testing Strategy

- Drivers
 - allows you to test a function without all the rest of a program
 - just to execute the function and show its results
 - often, provides a loop to retest the function on different arguments

Testing Strategy

- Stubs
 - simplified version of a function not written or tested yet
 - often used when testing another function
 - does not necessarily deliver correct values
 - works best when stubs are replaced by actual functions, one at a time

Summary

- Pointers
- Parameter Passing Mechanisms
- Problem Solving and Testing Strategy