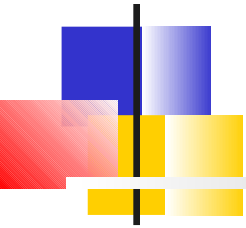
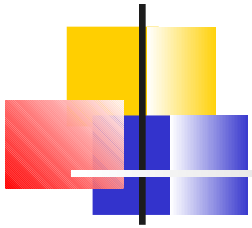


Pointers

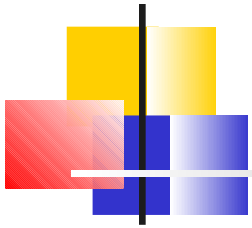
Chapter 11



Objectives

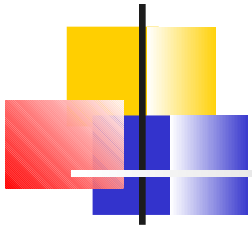
You will be able to

- Use pointers to access the contents of variables in a C program.
- Declare pointer variables correctly.
  - Initialize pointers in the declaration
- Assign values to pointer variables.
- Correctly compare pointers
  - and variables to which they point.



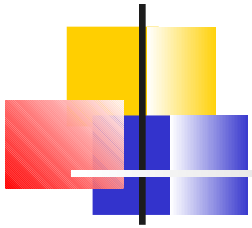
Pointers

- Pointers permit a program to work with *addresses* of variables rather than just their values.
  - Powerful feature of the C language
  - Also treacherous.
    - Easy to make mistakes
    - Mistakes can be very difficult to track down.
  - You have to be very careful with pointers!



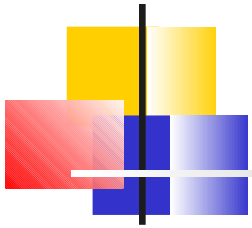
Why would we want to do this?

- Necessary in system programming
  - Operating systems, device drivers, etc.
- Permit a function to modify caller's non-array data
  - Like we are able to do with arrays
- Permit a function to return more than one value to the caller using parameters.
  - Can be of various types.
  - Can have meaningful names.



Why would we want to do this?

- Required for complex data structures
- Permit a function to access complex data structures in caller’s address space.



Pointer Variables

- **Pointer variables are specific to a type**

  int *pCount;

  - The * means that pCount is a *pointer* to an int.
  - It is a variable that may be used to hold the *address* of an int

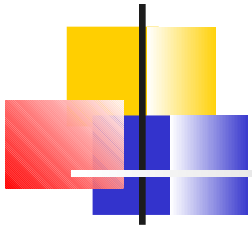
  Outside the declaration, *pCount is the name of the "pointee", i.e., the "target" of the pointer.

  The * in *pCount is the *dereferencing* operator.

- We use *pCount like a normal int variable.

  - But we have to be sure that pCount is set to point to an actual integer variable.

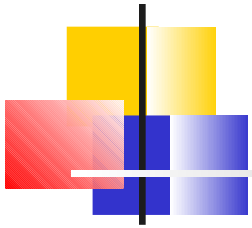
  - This does not happen automatically!



Setting Pointer Variables

- Pointer variables have to be set to the *address* of a something before being used.
- To get the address of a variable, put & in front of the name.
  - As for scanf
- Example

```
 int value;
int *pValue;
....
pValue = &value;
```

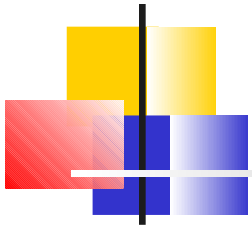


Using a Pointer

```
int main ()
{
    int value = 100;
    int *pValue;

    printf ("Before increment, value = %d\n", value );

    pValue = &value;
    *pValue += 1;

    printf ("After increment, value = %d\n", value );
    return 0;
}
```

Set pointer pValue to address of value

Increment the int variable that pValue points to



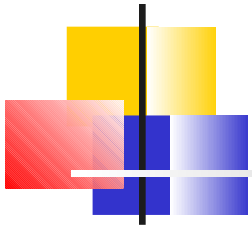
Dereferencing a Pointer

- Effectively replaces the pointer by the variable to which it points.
- If pValue is a pointer to an int, *pValue *is* an int
  - Equivalent to using the name of the variable whose address is in pValue.
  - * has very high precedence
    - like all unary operators.

Using a Pointer

turnerr@login2:~/test

```
/home/t/turnerr/test
[turnerr@login2 test]$ ls
ptr_inc.c
[turnerr@login2 test]$ cat ptr_inc.c
#include <stdio.h>

int main ()
{
    int value = 100;
    int * pValue;

    printf ("Before increment, value = %d\n", value);

    pValue = &value;
    *pValue += 1;

    printf ("After increment, value = %d\n", value);
    return 0;
}


[turnerr@login2 test]$ gcc -Wall ptr_inc.c
[turnerr@login2 test]$ ./a.out
Before increment, value = 100
After increment, value = 101
[turnerr@login2 test]$
```



Using a Pointer

- We could have used *pValue in the printf

```
#include <stdio.h>

int main ()
{
    int value = 100;
    int * pValue;

    pValue = &value;        // Set pointer before using it

    printf ("Before increment, value = %d\n", *pValue);

    *pValue += 1;

    printf ("After increment, value = %d\n", *pValue);
    return 0;
}
```

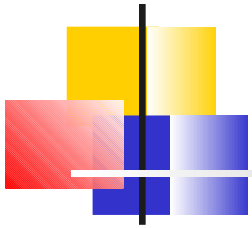
Same effect as value.

Using a Pointer

turnerr@login2:~/test

```
[turnerr@login2 test]$
[turnerr@login2 test]$
[turnerr@login2 test]$ cat ptr_inc2.c
#include <stdio.h>

int main ()
{
    int value = 100;
    int * pValue;

    pValue = &value;
    printf ("Before increment, value = %d\n", *pValue);

    *pValue += 1;

    printf ("After increment, value = %d\n", *pValue);
    return 0;
}


[turnerr@login2 test]$
[turnerr@login2 test]$ gcc -Wall ptr_inc2.c
[turnerr@login2 test]$ ./a.out
Before increment, value = 100
After increment, value = 101
[turnerr@login2 test]$
```

Same result.



Using a Pointer

Summary

If pValue is a pointer to int

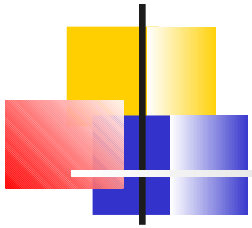
*pValue works like a normal int variable

- Use on either side of an assignment

```
*pValue = some_other_int;
some_other_int = *pValue;
```

- Use in function calls

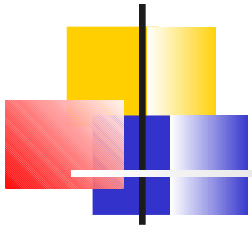
```
printf (“Value is %d\n”, *pValue);
```



Using a Pointer

- Pointers to other types work the same

```
double *pRadius;
char *pSomeChar;
```



Pointer Declarations

- You can declare multiple pointer variables on a single line.

```
int *pN1, *pN2, *pN3;
```

- This is a bad practice.
- Common source of errors in C programs.

A Source of Confusion

- Does the * bind to int or to pN1?

```
int *pN1, *pN2, *pN3;
```

- vs

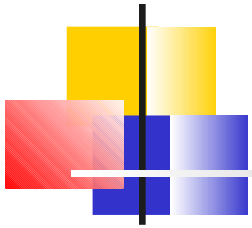
```
int* pN1, pN2, pN3;
```

These are int variables.

NOT pointers to int.

- Spaces before and after the * are not significant.

- The * binds to pN1, not to int.
- * says the name that follows is a pointer to the type that precedes it.



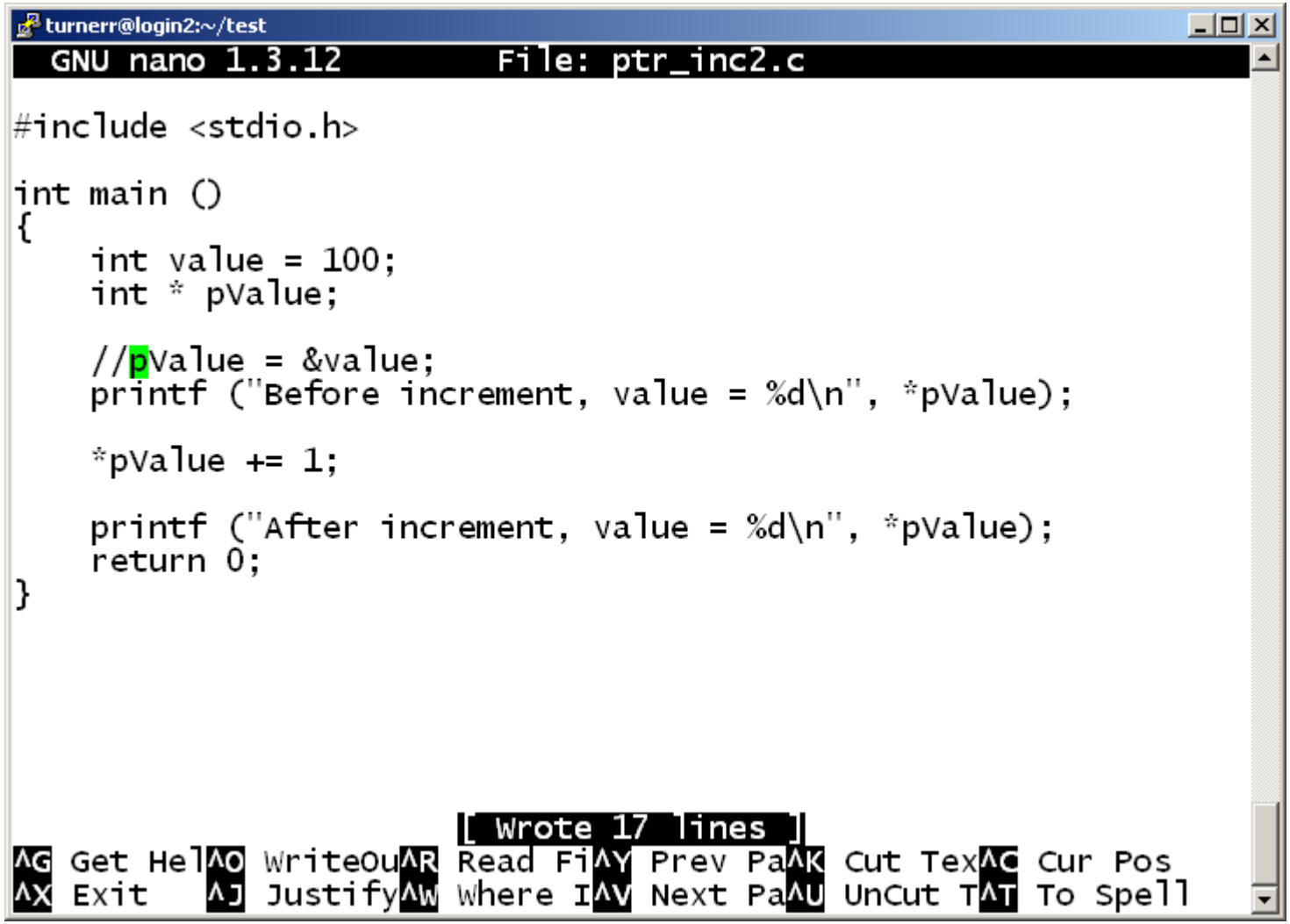
To Avoid Possible Confusion

- For this class, use a separate line for each pointer declaration.

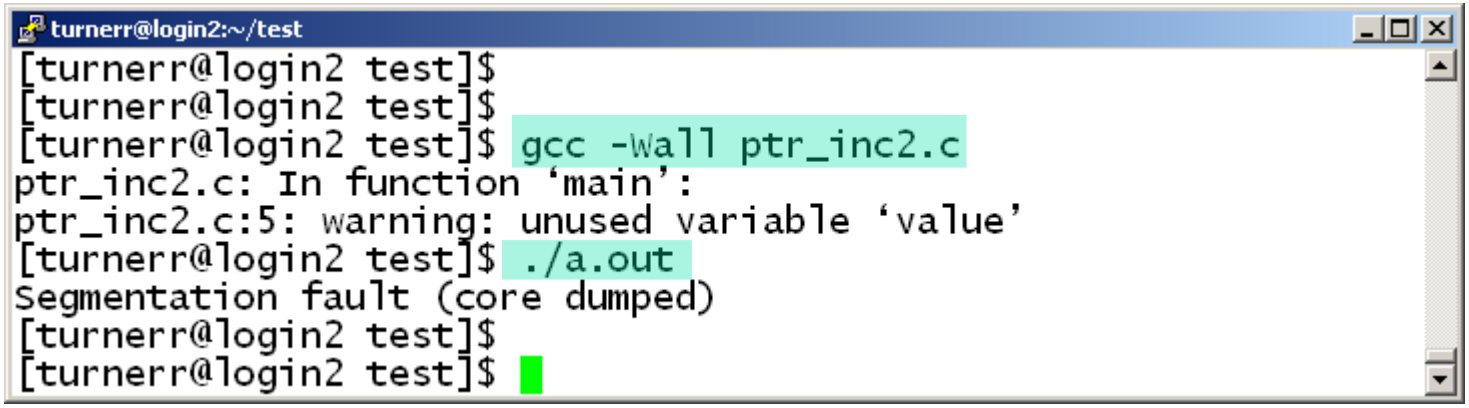
```
int *pN1;  /* Pointer to ... */
int *pN2;  /* Pointer to ... */
int *pN3;  /* Pointer to ... */
```

- Programming Style Guideline

Using an Uninitialized Pointer

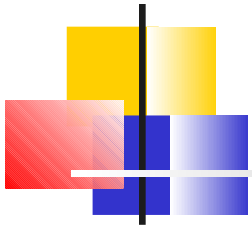


Using an Uninitialized Pointer



The effects of using an uninitialized pointer are *unpredictable.*

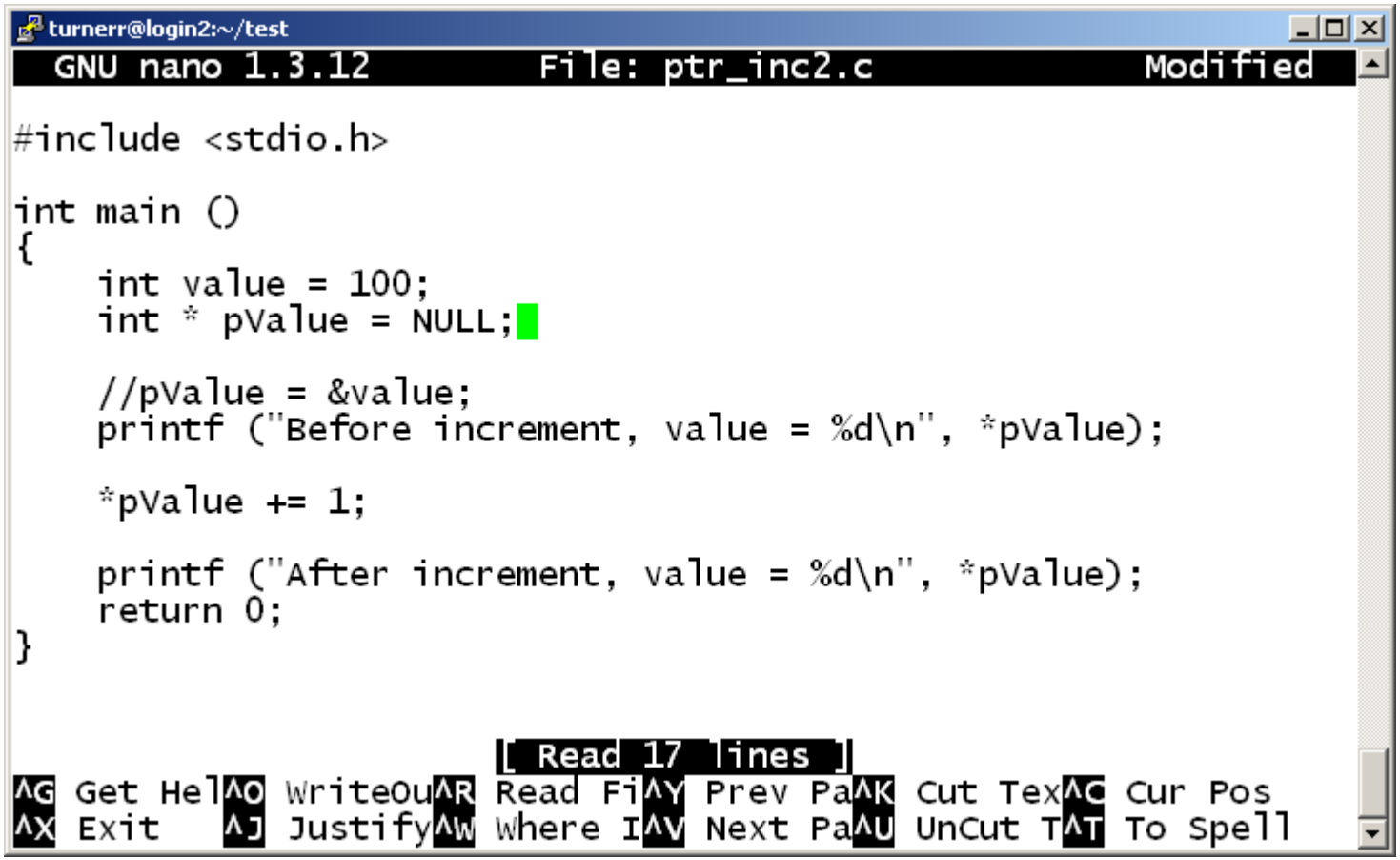
You might see something completely different.



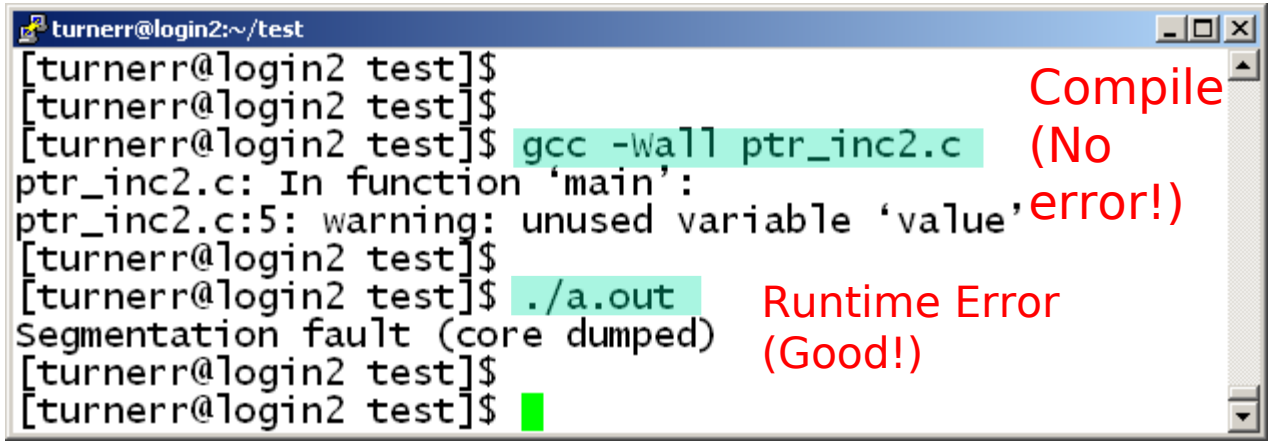
The NULL Pointer

- NULL is defined as address 0
  - An invalid address for program references
- Set pointers to NULL to show that they are not valid.
  - Results in immediate runtime error if used

Using a NULL Pointer



Using a NULL Pointer



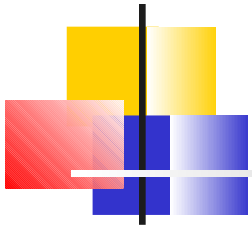
Initializing a Pointer in the Declaration

- You can provide a valid value for a pointer in the declaration.

```
int value = 100;
int *pValue = &value;
```

Looks like an assignment statement, but it's not.

Initializes pValue, NOT *pValue

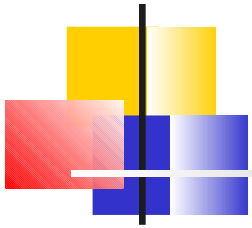


Initializing a Pointer in the Declaration

```
#include <stdio.h>

main ()
{
    int value = 100;
    int *pValue = &value;

    printf ("Before increment, value = %d\n", value);

    *pValue += 1;

    printf ("After increment, value = %d\n", value);
}
```

Initialize pValue to point to value

Increment the int variable that pValue points to.



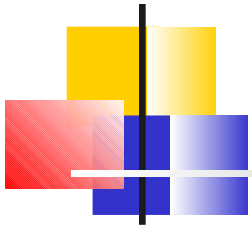
Defining Pointers

- New Programming Style Guideline:
  - Every pointer declaration should provide an initial value.
  - If you know the value at compile time, use it for initialization.

```
int * pValue = &Value;
```

  - If you don’t know the value at compile time initialize the pointer to NULL.

```
int * pValue = NULL;
```



Comparing Pointers

- What does it mean when you compare pointers?

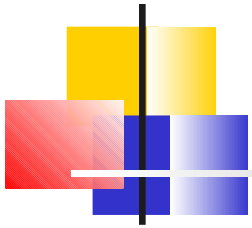
```
if (pValue1 == pValue2) ...
```

This asks if the pointers point to the same variable.

Are the pointers identical?

```
if (*pValue1 == *pValue2) ...
```

This asks if the *values that they point to* are identical



Exercise

Exercise 11.2 If i is a variable and p points to i, which of the following expressions are aliases for i:

(a) *p	**Alias for i**
(b) &p	**Not an alias for i; address of p**
(c) *&p	**Not an alias for i; alias for p**
(d) &*p	**Not an alias for i; address of i**
(e) *i	**Not an alias for i; illegal expression, i is not a pointer**
(f) &i	**Not an alias for i; address of i**
(g) *&i	**Alias for i**
(h) &*i	**Not an alias for i; illegal expression, i is not a pointer**