Problem Solving and Programming

Pointers

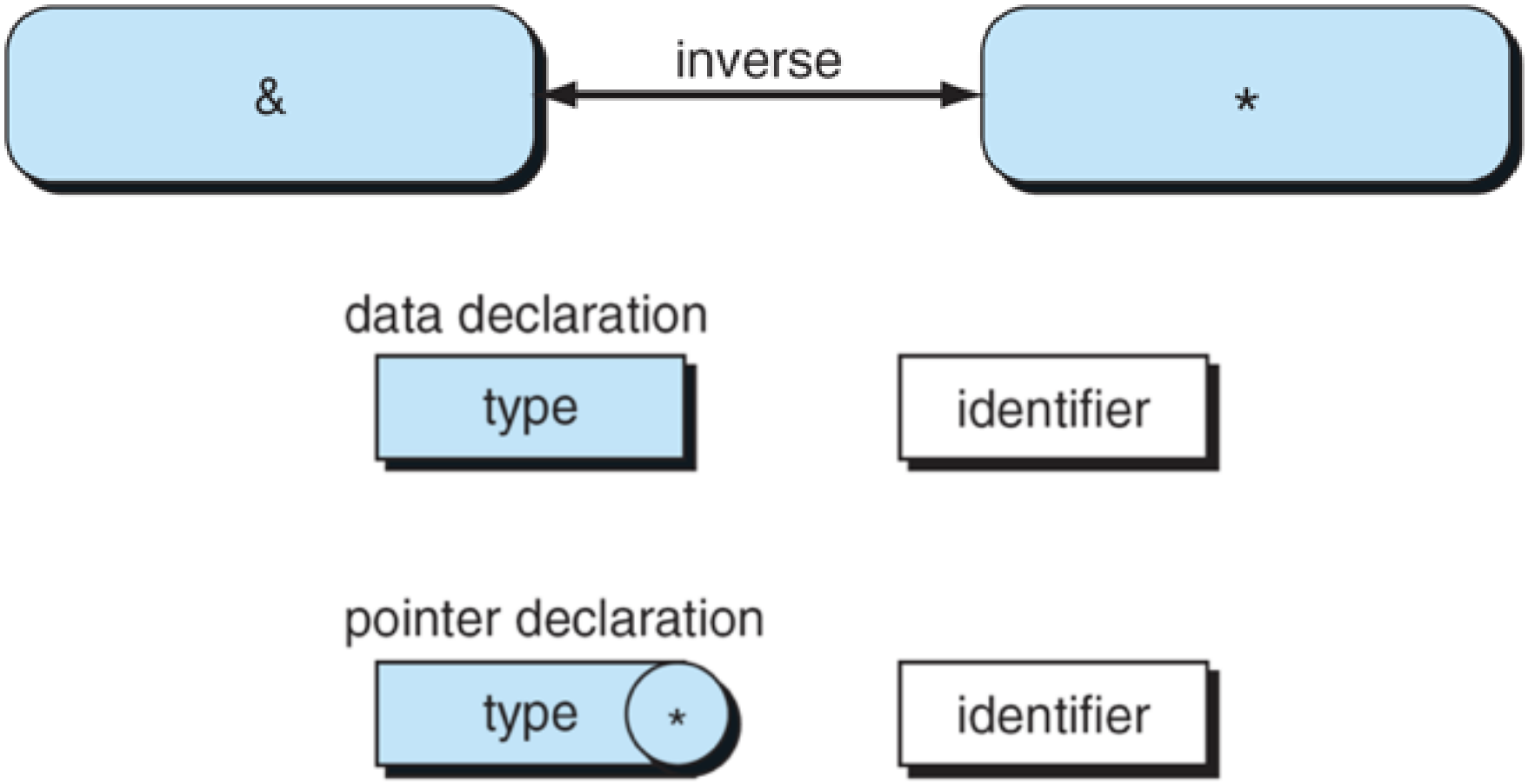
Kasun de Zoysa
kasun@ucsc.cmb.ac.lk

UNIVERSITY OF COLOMBO SCHOOL OF COMPUTING

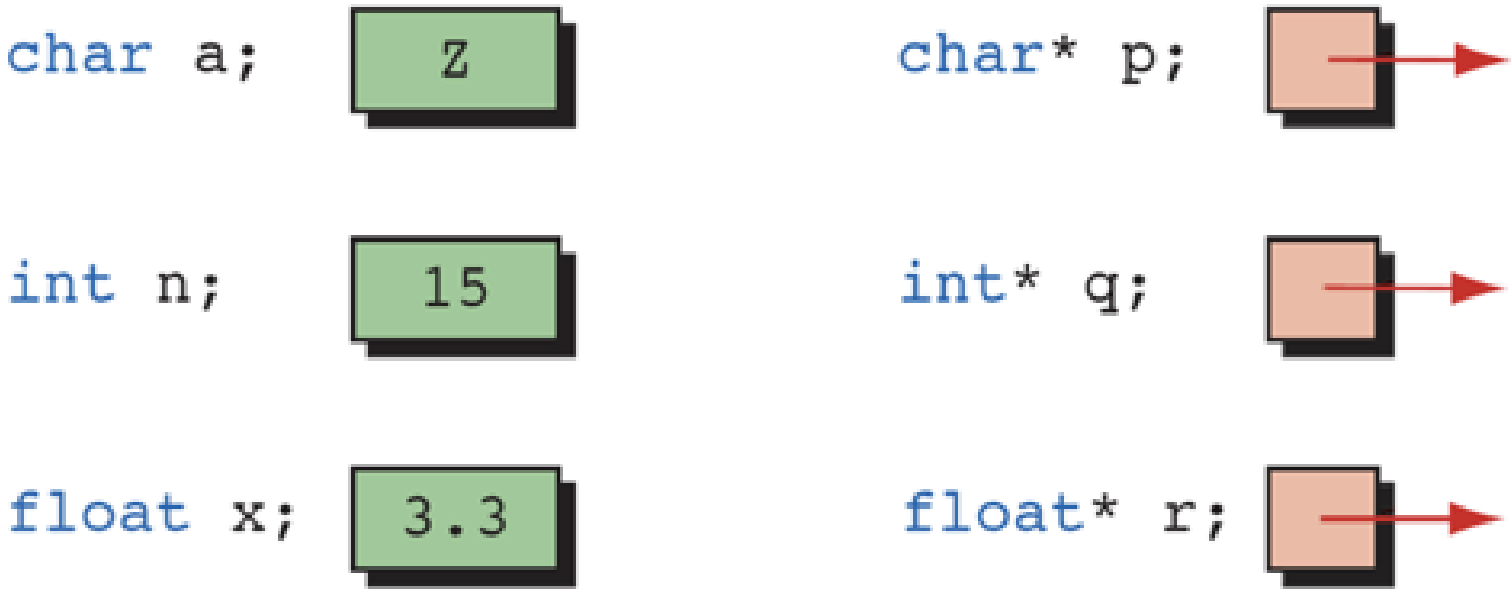
Pointers

A pointer is a constant or variable that contains an address that can be used to access data.

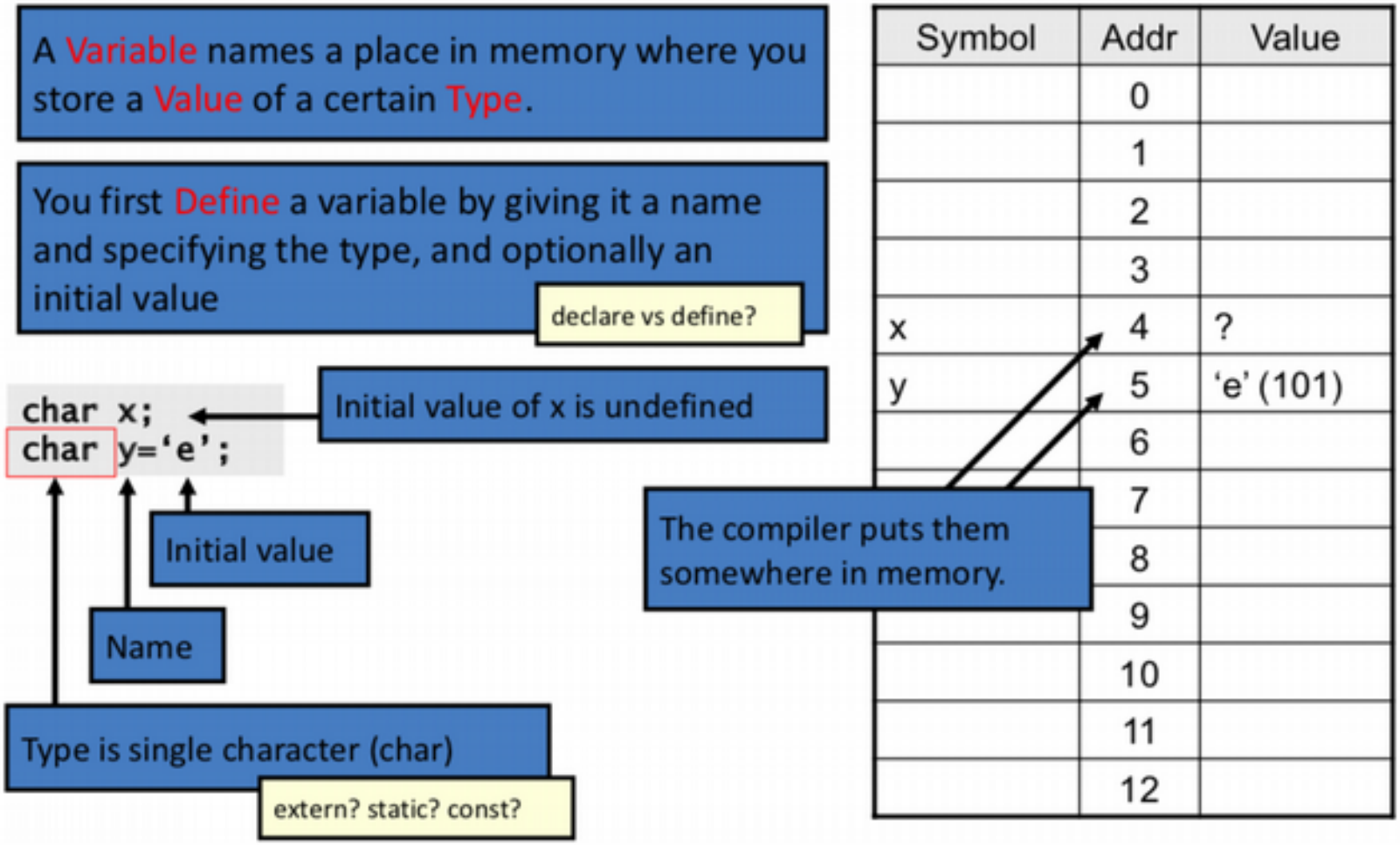
Address and Indirection Operators



Declaring Pointer Variables



Pointers


A Variable names a place in memory where you store a Value of a certain Type.
You first Define a variable by giving it a name and specifying the type, and optionally an initial value
declare vs define?
char x;
char y='e';
Initial value of x is undefined
Initial value
Name
Type is single character (char)
extern? static? const?
The compiler puts them somewhere in memory.
Symbol
Addr
Value
0
1
2
3
x
4
?
y
5
'e' (101)
6
7
8
9
10
11
12

Pointers

This is exactly how "pointers" work.

```
"address of" or reference operator:  &
"memory_at" or dereference operator: *
```

```
void f(address_of_char p)
{
  memory_at[p] = memory_at[p] - 32;
}
```

```
char y = 101;       /* y is 101 */
f(address_of(y));   /* i.e. f(5) */
/* y is now 101-32 = 69 */
```

A "pointer type": pointer to char

```
void f(char * p)
{
  *p = *p - 32;
}
```

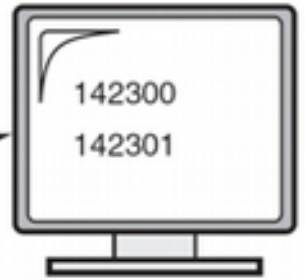
```
char y = 101;       /* y is 101 */
f(&y);              /* i.e. f(5) */
/* y is now 101-32 = 69 */
```

Pointers are used in C for many other purposes:

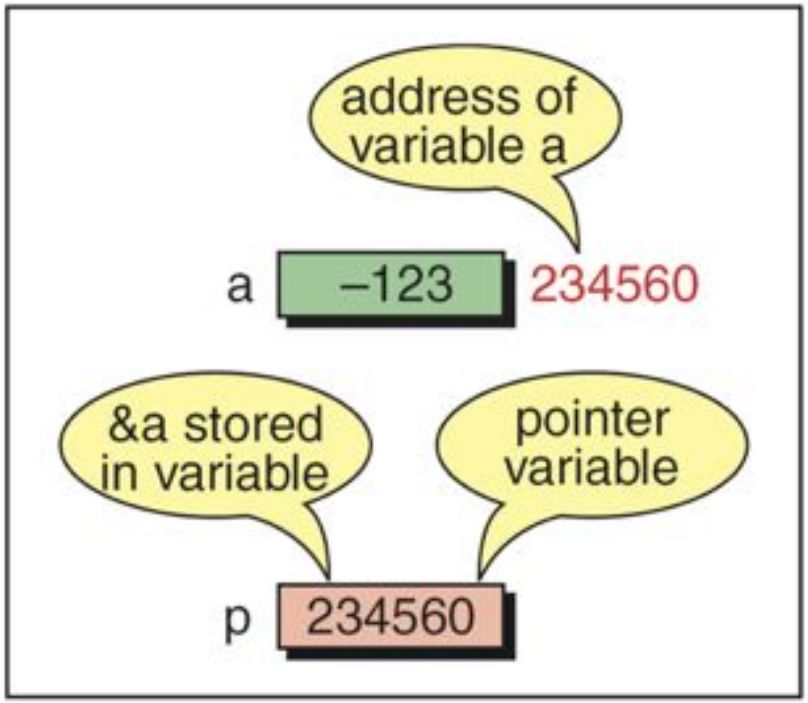
- Passing large objects without copying them
- Accessing dynamically allocated memory
- Referring to functions

Print Character Addresses

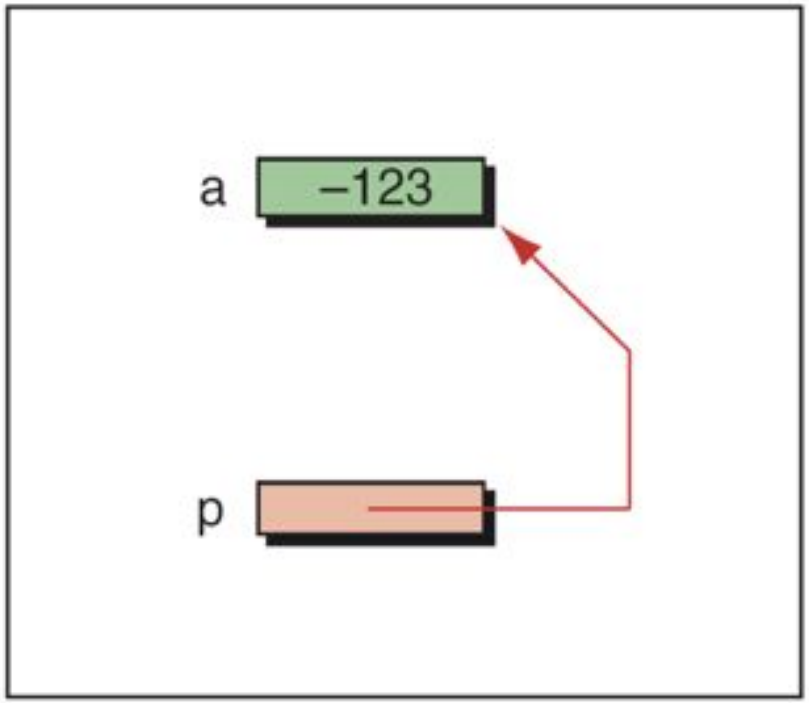
```
// Print character addesses
#include <stdio.h>

int main (void)
{
// Local Declarations
   char a;
   char b;
// Statements
   printf ("%p\n %p\n", &a, &b);
   return 0;
}  // main
```



Pointer Variable

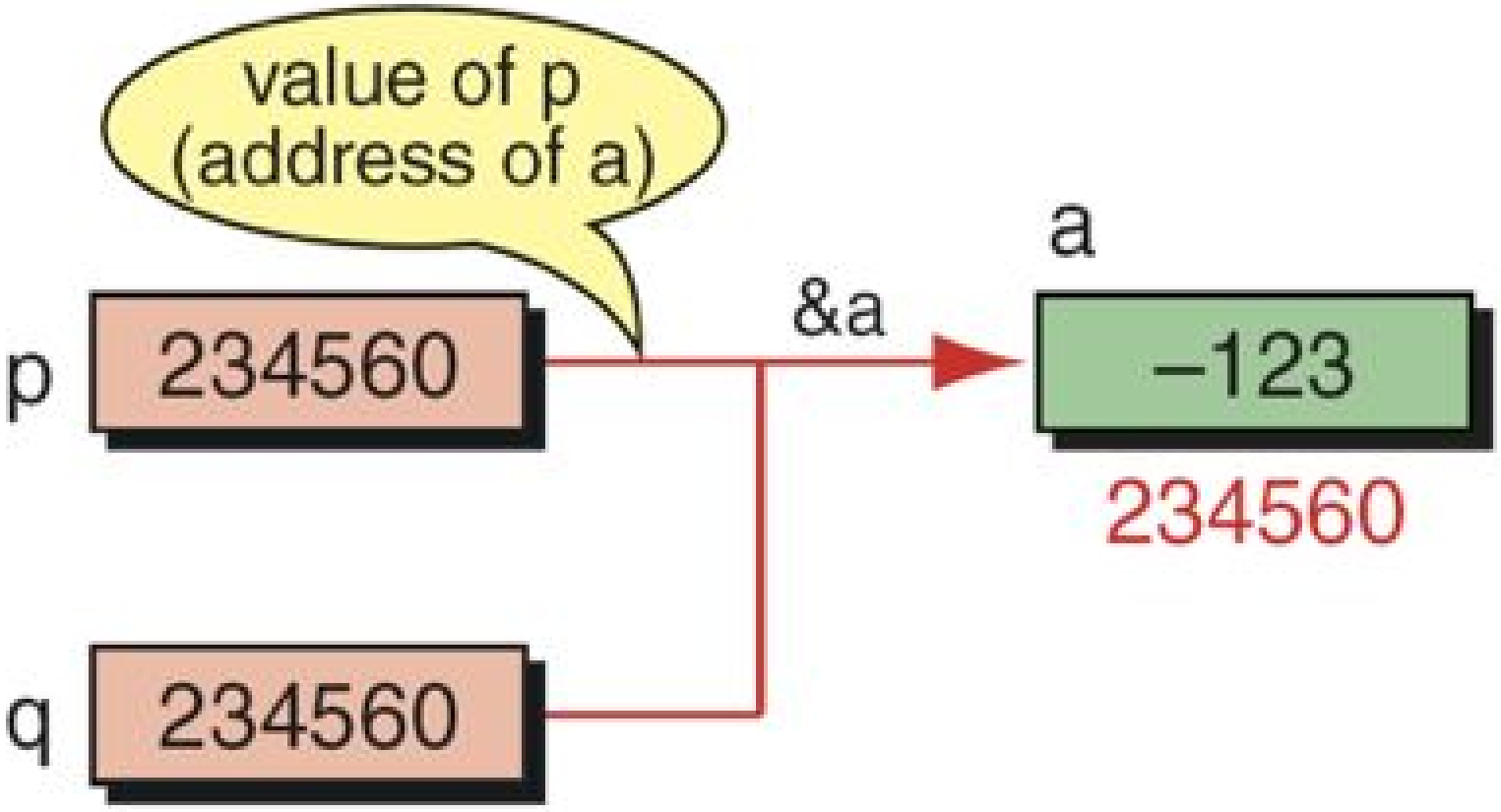


Physical representation



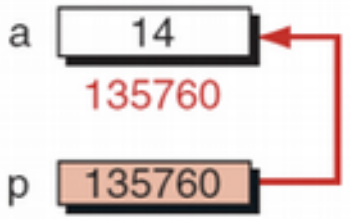
Logical representation

Multiple Pointers to a Variable

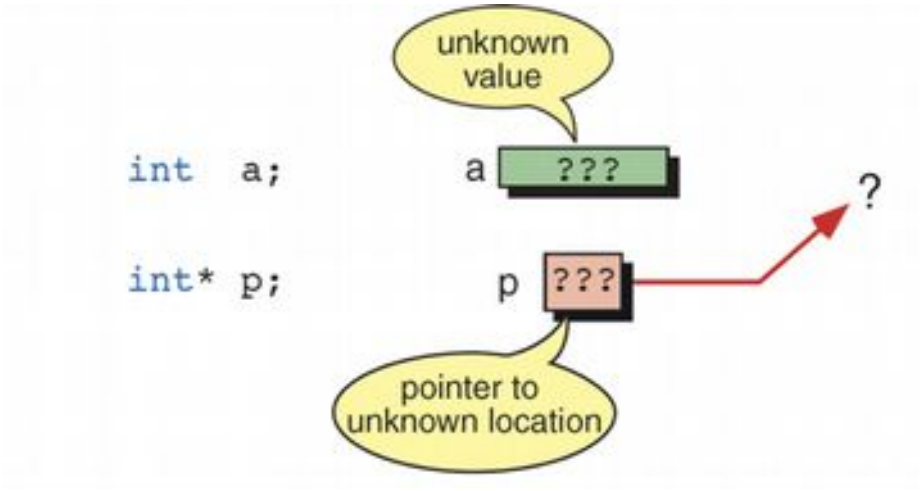


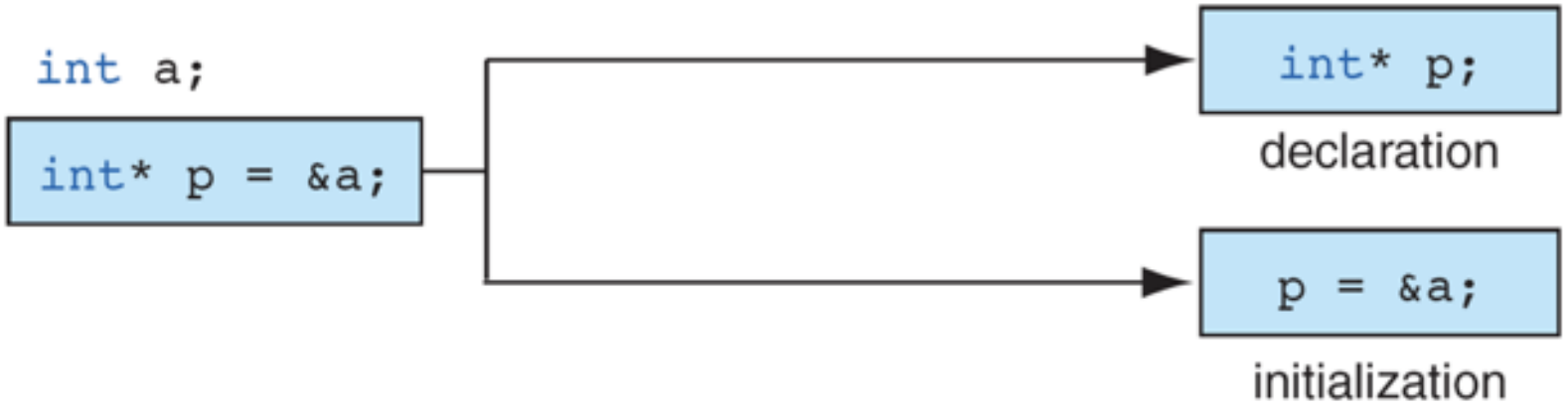
Use of Pointers

```
1  /* Demonstrate pointer use
2        Written by:
3        Date:
4  */
5  #include <stdio.h>
6
7  int main (void)
8  {
9  // Local Declarations
10    int  a;
11    int* p;
12
13 // Statements
14    a = 14;
15    p = &a;
16
17    printf("%d %p\n", a, &a);
```

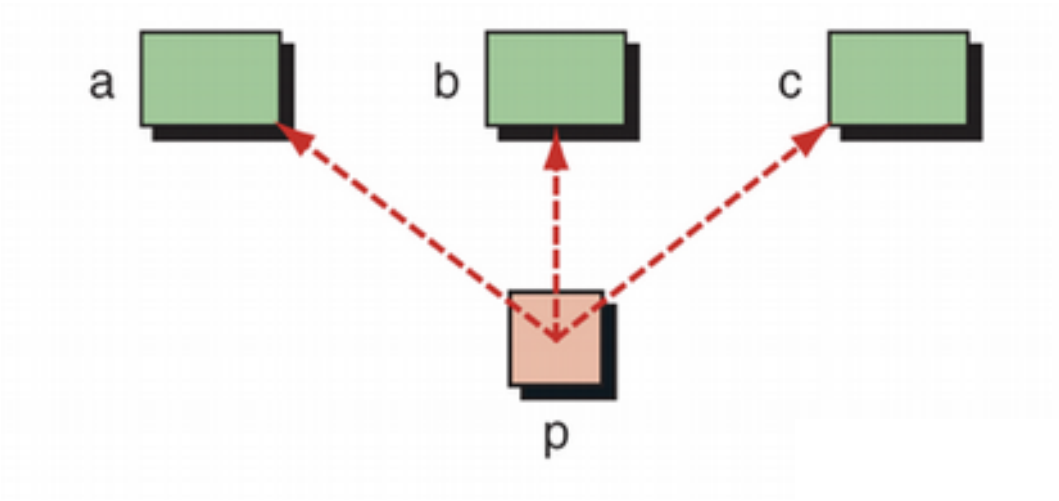


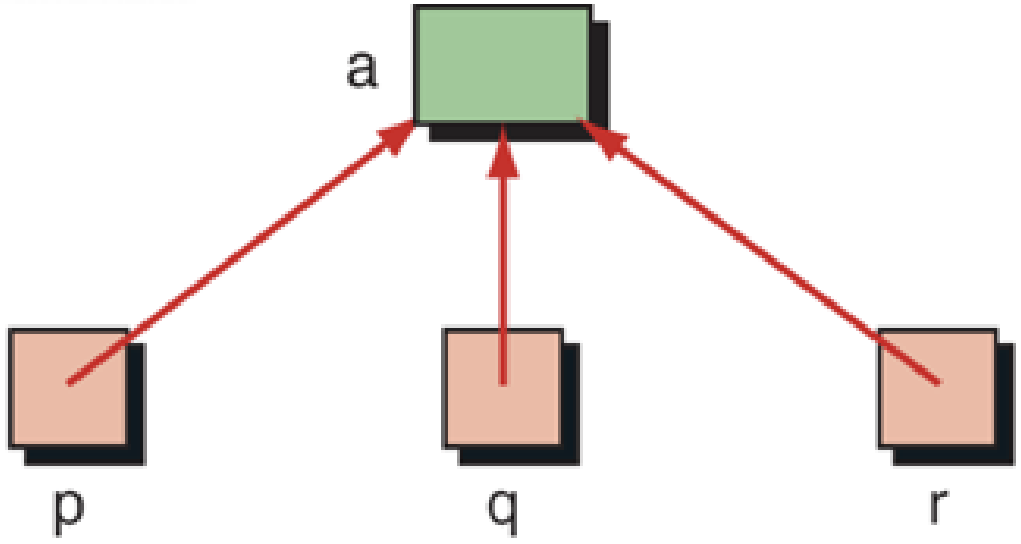
Initializing Pointer Variables





Variables and Pointers





Swaps

The program swaps the actual variables values because the function accesses them by address using pointers.

```
void swaps(int* x,int* y)
{  int p=*x;
 *x=*y; *y=p;
 }
int main(){
 int x=2,y=5;
 swaps(&x,&y);
 return 0;
}
```

Functions with Array Parameters

In C, we cannot pass an array by value to a function. Whereas, an array name is a pointer (address), so we just pass an array name to a function which means to pass a pointer to the array.

```
int main(int argc,char *argv[]){
 printf("Number of parameters = %d \n",argc);
 for(int i=0;i<argc;i++) puts(argv[i]);
 return 0;
}
```

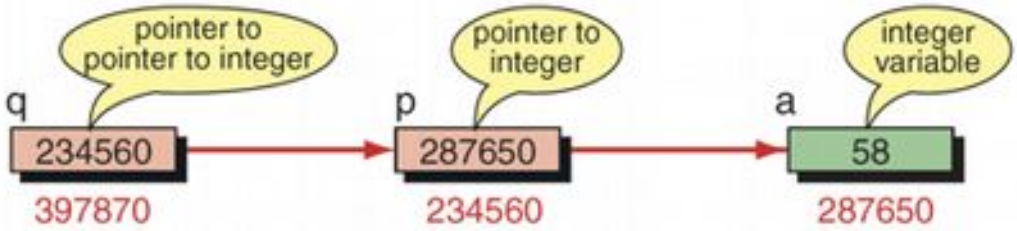
Pointers to Pointers

So far, all our pointers have been pointing directly to data.

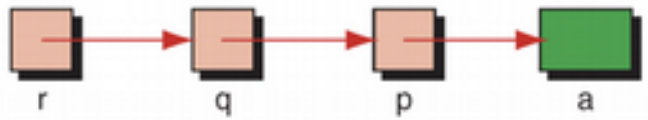
It is possible and with advanced data structures often necessary to use pointers that point to other pointers. For example, we can have a pointer pointing to a pointer to an integer.

Pointers to Pointers

```
// Local Declarations
int   a;
int*  p;
int** q;
```




```
// Statements
a = 58;
p = &a;
q = &p;
printf(" %3d",   a);
printf(" %3d",  *p);
printf(" %3d", **q);
```



Function Pointer

In C, like normal data pointers (int *, char *, etc), we can have pointers to functions. Following is a simple example that shows declaration and function call using function pointer.

```c
#include <stdio.h>
// A normal function with an int parameter
// and void return type
void fun(int a)
{
    printf("Value of a is %d\n", a);
}

int main()
{
    // fun_ptr is a pointer to function fun()
    void (*fun_ptr)(int) = &fun;

    /* The above line is equivalent of following two
       void (*fun_ptr)(int);
       fun_ptr = &fun;
    */

    // Invoking fun() using fun_ptr
    (*fun_ptr)(10);

    return 0;
}
```

Function Pointer

```
int main(int argc, char *argv[]){
  int x=0,a=5,b=10;
  int (* p)(int,int);
  if(argv[1][0]=='+') p=&f1;
    else if (argv[1][0]=='-') p=&f2;
      else printf("Wrong Op\n");
  x=(*p)(a,b);
 printf("X=%d\n",x);

 return 0;
}
```

GNU Debugger (gdb)

It allows you to inspect what the program is doing at a certain point during execution.

Errors like segmentation faults may be easier to find with the help of gdb.

Normally, you would compile a program like:

```
gcc [flags] <source files> -o <output file>
```

Now you add a -g option to enable built-in debugging support (which gdb needs):

```
gcc [other flags] -g <source files> -o <output file>
```

Starting up gdb

Just try "gdb" or "gdb prog1.x." You'll get a prompt that looks like this:

```
(gdb)
```

If you didn't specify a program to debug, you'll have to load it in now:

```
(gdb) file prog1.x
```

Here, prog1.x is the program you want to load, and "file" is the command to load it.

Running the program

To run the program, just use:

```
(gdb) run
```

This runs the program.

- If it has no serious problems (i.e. the normal program didn't get a segmentation fault, etc.), the program should run fine here too.
- If the program *did* have issues, then you (should) get some useful information like the line number where it crashed, and parameters to the function that caused the error:

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000000400524 in sum_array_region (arr=0x7fffc902a270, r1=2, c1=5,
r2=4, c2=6) at sum-array-region2.c:12
```

Setting the breakpoint

Breakpoints can be used to stop the program run in the middle, at a designated point. The simplest way is the command "`break`." This sets a breakpoint at a specified file-line pair:

```
(gdb) break file1.c:6
```

This sets a breakpoint at line 6, of `file1.c`. Now, **if** the program ever reaches that location when running, the program will pause and prompt you for another command.

Tip

You can set as many breakpoints as you want, and the program should stop execution if it reaches any of them.

Now what?

- Once you've set a breakpoint, you can try using the `run` command again. This time, it should stop where you tell it to (unless a fatal error occurs before reaching that point).
- You can proceed onto the next breakpoint by typing "`continue`" (Typing `run` again would restart the program from the beginning, which isn't very useful.)

```
(gdb) continue
```

- You can single-step (execute *just* the next line of code) by typing "`step`." This gives you really fine-grained control over how the program proceeds. You can do this a *lot*...

```
(gdb) step
```

Printing the value

- So far you've learned how to interrupt program flow at fixed, specified points, and how to continue stepping line-by-line. However, sooner or later you're going to want to see things like *the values of variables*, etc. This *might* be useful in debugging. :)
- The `print` command prints the value of the variable specified, and `print/x` prints the value in hexadecimal:

```
(gdb) print my_var
(gdb) print/x my_var
```

Watching the variable

Whereas breakpoints interrupt the program at a particular line or function, watchpoints act on variables. They pause the program whenever a *watched* variable's value is modified. For example, the following `watch` command:

```
(gdb) watch my_var
```

Now, whenever `my_var`'s value is modified, the program will interrupt and print out the old and new values.

Other useful commands

- `backtrace` - produces a stack trace of the function calls that lead to a seg fault (should remind you of Java exceptions)
- `where` - same as `backtrace`; you can think of this version as working even when you're still in the middle of the program
- `finish` - runs until the current function is finished
- `delete` - deletes a specified breakpoint
- `info breakpoints` - shows information about all declared breakpoints

Discussion