CS102: Pointers

Some novice programmers consider pointers to be the most confusing topic when learning to program in C. This is partly because pointers do not exist in several other popular languages, and partly because many programmers don't really like to think about what is going on in memory when they are using variables. Still, there are also some subtle aspects of pointers that are genuinely confusing.

A **pointer** is a variable whose value represents a memory address. The memory address it "points to" (store the memory address of) will store a value of a certain type. Often, a pointer will point to another variable.

Remember that every variable (including a pointer variable) is a named memory address. Each variable has a value and a type. The type of the variable determines, among other things, the number of bytes in memory set aside for the variable.

Let's make some more sense of it with an example that will introduce several concepts:

```
#include <stdio.h>

int main(void)
{
  int x;
  int *p;

  x = 5;
  p = &x;
  printf("The value of x is %d.\n", x);
  printf("The address of x is %p.\n", &x);
  printf("The value of p is %p.\n", p);
  printf("The value that p points to is %d.\n", *p);
  printf("The address of p is %p.\n", &p);

  return 0;
}
```

At one time, when I ran this program on a particular computer, I saw the following output:

```
The value of x is 5.
The address of x is 0x28ac5c.
The value of p is 0x28ac5c.
The value that p points to is 5.
The address of p is 0x28ac58.
```

The 5's will be the same every time you run the program. The other values will change, but the second and third will always be the same as each other.

This program declares the variable "p" to be a pointer to an integer. The asterisk ('*') used in a declaration means that the variable following it is a pointer to a value of the type indicated to the left of the asterisk. In other words, "p" will store a memory address where an integer value is stored.

Next, we are going to look at the **address-of operator** ('&'). We've seen it before when we used "scanf". The '&' applied to the left of a variable obtains the address of that variable. The line "p = &x" assigns the memory address of the variable "x" to the pointer "p".

Now let's look at the first three "printf" statements. The first is something we've seen many times; we are printing out the value of the integer variable "x", and the value is 5. The second is something new. We are printing out the memory address where the variable "x" is stored in memory. It is important to use the "%p" field specification when printing out memory addresses, since on some systems integers are not the same size as memory addresses.

The "%p" field specification causes the memory address to be printed out in hexadecimal format. Hexadecimal is base 16. The digits in hexadecimal are 0 through 9 and A through F (representing digits 10 through 15). The right most digit in a hexadecimal number is the one's digit, the next digit to the left is the 16's digit, the next digit over is the 256's digit, etc. The reason that hexadecimal is used commonly when discussing computers is that 16 is a power of 2 (2^4), and so each hexadecimal digit is represented by 4 bits (half a byte). Every byte can therefore be represented by two hexadecimal digits without any waste.

The third "printf" statement prints out the value of "p". But "p" is a pointer, and its value is the memory address of "x". This is why the third "printf" statement prints out the same thing as the second "printf" statement.

Now let's look at the fourth "printf" statement. We've seen that when an asterisk is used in a declaration, it is used to declare a variable as a pointer. When used elsewhere, the asterisk is the **indirection operator**. It obtains the value stored at the memory address located to the right of the operator. In this case, the memory address is represented by the variable "p". The value stored at that memory address is the integer 5. This explains the fourth printf statement (sort of, we'll explain more a bit later). When you access the value a pointer points to, this is known as **dereferencing** the pointer.

It is common terminology to say that a pointer "points to" the memory address that it stores. You often see it pictorially represented with arrows.

Now, let's look at the last "printf" statement. The pointer "p" is also a variable, and its value has to be stored in memory somewhere. The last "printf" is printing out the memory address where "p" itself is stored. If you run this program, you will not likely see the same values as above, but local variables within a function are stored close together in memory, so the memory address of "p" will be close to the memory address of "x".

We have been talking about memory addresses of integer variables and integers stored at a given memory address. But on most modern systems, integers require four bytes of memory! The address of a variable (or value) is actually the memory address of the first byte occupied by that variable (or value).

So, in the program above, "x" probably occupies four bytes of memory. The first of those bytes is located at the value that was displayed.

Looking back at the fourth "printf" statement, when we printed out what "p" points to (using the indirection operator), how did the computer know to use all four bytes? Because it remembers that "p" was declared as a pointer to an integer, and integers require four bytes. Additionally, the "%d" in the "printf" format string is telling the "printf" function how we want the value to be displayed (as an integer).

Sometimes, you want a pointer to point to nothing. For this purpose, C supplies the NULL constant, which is defined in the standard input and output library. If you try to use the indirection operator to the left of a NULL memory address, your program will almost certainly crash! For example, this program will probably crash:

```
#include <stdio.h>

int main(void)
{
  int *p;

  p = NULL;
  printf("The value that p points to is %d.\n", *p);

  return 0;
}
```

A crash would also occur if you try to assign a value to memory pointed to by a NULL pointer. For example:

```
p = NULL;
*p = 5;
```

will cause a crash. Either way, this type of error is known as dereferencing a NULL pointer. It is one of the most common causes of crashes in C programs.

As with all other variables, pointers are not initialized automatically. The following program may crash, or it may print out a random value, depending on whether or not the value that happens to be in “p” represents a valid memory address.

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

```
int main(void)
{
  int *p;

  printf("The value that p points to is %d.\n", *p);

  return 0;
}
```

This is probably a good time to note that when your C program crashes, on some systems, it leaves a file called "core" in your current directory, or on some systems, it leaves a file ending with ".stackdump". This file, if it is created, contains information about the state of your program when it crashed, and it is possible to use it for debugging purposes, but normally, you just want to delete it. (To remove a file on a Cygwin, Linux, or Unix system, use the "rm" command at the prompt. For example, "rm a.exe.stackdump" will get rid of the file named "a.exe.stackdump".)

It is possible to initialize a pointer along with its declaration. For example:

```
int *p1 = &x;
```

This declares a pointer named "p1" to be a pointer to integer, and it initializes it with the memory address of the variable "x" (which must have been previously declared to be an integer variable). You can also declare multiple pointers with one declaration, and initialize one or more of them. For example:

```
int *p1 = &x, *p2, *p3 = &y;
```

This declares three pointers and initializes two of them. Note that each pointer must have its own asterisk in the declaration!

You can mix declarations of pointers with normal variables, but this can get confusing. For example:

```
int *p1, p2, p3;
```

You might think that you are declaring three pointers here, but, in fact, this declares just one pointer ("p1") and two normal integer variables ("p2" and "p3"). To avoid this type of confusion, I think it is usually best to declare pointers separately from non-pointers.

It is also possible to assign one pointer to another. For example, if "p1" and "p2" are both pointers to integers, you can say:

```
p1 = p2;
```

Whatever memory address is stored by "p2" will be assigned to "p1" (i.e., they will both point to the same memory location).

Here is another example:

```
#include <stdio.h>

int main(void)
{
  int x = 5;
  int *p1 = &x, *p2 = &x;

  printf("The value of x is %d.\n", x);
  printf("The value that p1 points to is %d.\n", *p1);
  printf("The value that p2 points to is %d.\n", *p2);

  *p1 = 10;

  printf("The value of x is %d.\n", x);
  printf("The value that p1 points to is %d.\n", *p1);
  printf("The value that p2 points to is %d.\n", *p2);

  return 0;
}
```

This program displays the following:

```
The value of x is 5.
The value that p1 points to is 5.
The value that p2 points to is 5.
The value of x is 10.
The value that p1 points to is 10.
The value that p2 points to is 10.
```

OK, so what's going on here? The variable "x" is an integer variable. Remember, all variables are named memory locations, so somewhere in memory that is designated with the name "x" resides the value of 5. The variables "p1" and "p2" are pointers. They are still variables (named memory locations themselves), and they each store the address of the variable "x". They can be thought of as pointing to "x". Now consider the statement "*p1 = 10;". The asterisk to the left of the variable "p1" dereferences the pointer, and the expression "*p1" refers to the memory address referenced by the pointer. We can use this to the left of an equal sign (the assignment operator) to reassign this value, and we are changing it to 10. (Technically, we can say that "*p1" is a valid l-value, meaning that it can appear to the left of an assignment operator.) Since "p2" points to the same memory, and "x" is just a name for that memory location, the three following "printf" statements are all referring to this new value.

If "p" is a pointer to an integer, the expression "*p" represents the integer itself, and this expression can be used the same way an integer variable can be used. In addition to lines

like "*p = 10", we could say "(*p)++", or even "x = (*p)++". The parentheses around the "*p" are necessary if we want to increment the value that "p" points to; otherwise, if we just wrote something like "*p++", it turns out that the increment operator has a higher precedence than the indirection operator and it would apply first. So you would actually be incrementing the value of the pointer "p" (which is a memory address), and then looking at what the new memory address points to (because of the indirection operator). Sometimes you do want to manipulate pointers in this way, but here we are just trying to increment that value pointed to by the pointer.

Why are pointers useful?

One reason that pointers are useful is that they provide a way for a programmer to allow functions to change the values of variables.

Let's look at an example that could have been used for the Tic-Tac-Toe program. In that program, we often want to convert a square (a number from 1 to 9) to a row and column, and we've seen that we can do this with the two lines:

```
row = (square - 1) / 3;
column = (square - 1) % 3;
```

Let's say we need to do this many times and want to do it with a function. Since each function can only return one value, the only way we can do it without pointers is to create two separate functions, one that is passed the square and returns the row, and another that is passed the square and returns the column. It generally doesn't pay to write a function to replace a single line of code, so we don’t even bother. However, with pointers, we can have the following function:

```
void row_col_from_square(int square, int *row, int *col)
{
  *row = (square - 1) / 3;
  *col = (square - 1) % 3;

  return;
}
```

The prototype statement for this function would be:

```
void row_col_from_square(int, int *, int *);
```

This function gets passed one normal integer (an integer from 1 to 9 representing a square) and it also gets passed two pointers to integers. Remember, a pointer stores a memory address, and in this case we will be passing the function the address of variables whose values we want to change.

So let's say in our code we have the variable "square" set, and two other variables "row" and "column" we want to fill in. (Notice it doesn't matter if the names are the same as the

ones used in the function. I'm using an example where two of the three variables have the same name and one is different.) Instead of using two statements like we have been doing to calculate row and column, we can just call the function as follows:

```
row_col_from_square(square, &row, &column);
```

So we pass the function the value of "square" and the addresses of the two variables to which we want to assign values. The function will fill in the values of "row" and "column" for us. Nothing is returned.

Here is another example. Let's say you are writing a program in which it is commonly necessary to swap, or exchange, the value of two integer variables (not necessarily the same two each time). This actually requires three lines of code. For example, if you want to exchange the values of variables "x" and "y", and there is another variable of the same type named "temp", you can use:

```
temp = x;
x = y;
y = temp;
```

Note that you can NOT just use:

```
x = y;
y = x;
```

After these two statements, both variables would have the original value of "y".

Rather than use the three lines of code each time, let's say you want to write a function to swap two integer values. Without pointers, this cannot be done, but with pointers, it is simple:

```
#include <stdio.h>

void swap(int *, int *);

int main(void)
{
  int a = 5, b = 10;

  printf("Before swap: a = %d, b = %d.\n", a, b);
  swap(&a, &b);
  printf("After swap: a = %d, b = %d.\n", a, b);

  return 0;
}

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

```
{
  int temp;

  temp = *x;
  *x = *y;
  *y = temp;

  return;
}
```

This program will print:

```
Before swap: a = 5, b = 10.
After swap: a = 10, b = 5.
```

We are passing the addresses of variables "a" and "b" from "main" into the pointer parameters "x" and "y" used by the function "swap". Although functions can not directly change the values of variables passed to them (since they get sent a copy of the values), they can change the contents of memory! Since "a" and "b" are just named memory locations, if a function writes new values into those memory locations, the values of "a" and "b" are changed for good.

Perhaps you understand a little better now why you need to include the address-of operator ('&') to the left of all variables passed to the "scanf" function. As mentioned in earlier lectures, if you just pass a normal variable, like an integer or character, the called function cannot change the value of it. However, by passing the memory address of the variables into pointer parameters, the "scanf" function can write into the memory addresses that are represented by these pointers, thus changing the values of the original variables.

Many programmers think that if a function changes the value of a pointer that was passed to a function, it will stay changed after the function ends. This is not true. Functions can permanently change the value that the pointer points to, but not the value of the pointer itself (i.e., the memory address that it stores). This is best illustrated with an example:

```
#include <stdio.h>

void try_change(int *);

int x = 5;
int y = 10;

int main(void)
{
  int *p = &x;
  printf("Before function, p points to value %d.\n", *p);
  try_change(p);
```

```
  printf("After function, p points to value %d.\n", *p);

  return 0;
}

void try_change(int *p)
{
  p = &y;
  printf("In function, p points to value %d.\n", *p);
  return;
}
```

This program will display the following:

```
Before function, p points to value 5.
In function, p points to value 10.
After function, p points to value 5.
```

What is happening? The value of the pointer "p" is passed to "try_change". This value (which happens to represent the memory address of the global "x") is copied into the pointer "p" used by "try_change", but this is a different pointer "p" than the one used by "main". The pointer "p" visible to "try_change" is changed to point to the global "y" (i.e., it is assigned the memory address of "y"), explaining the second "printf". Then control returns to "main", but that pointer "p" has not been changed, explaining the third "printf".

Not all pointers point to integers. Below is an example of a program using a version of "swap" that handles float variables.

```
#include <stdio.h>

void swap(float *, float *);

int main(void)
{
  float a = 5.39, b = 10.21;

  printf("Before swap: a = %.2f, b = %.2f.\n", a, b);
  swap(&a, &b);
  printf("After swap: a = %.2f, b = %.2f.\n", a, b);

  return 0;
}

void swap(float *x, float *y)
{
  float temp;
```

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

  return;
}
```

This program will print:

```
Before swap: a = 5.39, b = 10.21.
After swap: a = 10.21, b = 5.39.
```

Another use of pointers involves the use of pointers to walk through the elements of an array. Arrays and pointers are treated very similar in C. In fact, the name of an array is a pointer constant to the first element of the array! The name of the array is a pointer to the first element of the array, but the value of this variable (the memory address that it points to) can not be changed. If a line of code tries to change the value of the name of an array, you will get a compiler error.

This might be clarified with the following example, which also introduces a few additional concepts:

```
#include <stdio.h>

int main(void)
{
  int a[5] = {10, 20, 30, 40, 50};
  int *p1;
  int *p2;

  p1 = a;
  p2 = &a[0];
  printf("a = %p, p1 = %p, p2 = %p.\n", a, p1, p2);
  printf("a[0] = %d, p1[0] = %d, p2[0] = %d.\n", a[0],
p1[0], p2[0]);
  printf("a[4] = %d, p1[4] = %d, p2[4] = %d.\n", a[4],
p1[4], p2[4]);
  printf("*a = %d, *p1 = %d, *p2 = %d.\n", *a, *p1, *p2);
  printf("*(a + 4) = %d, *(p1 + 4) = %d, *(p2 + 4) =
%d.\n", *(a + 4), *(p1 + 4), *(p2 + 4));

  return 0;
}
```

This program might display the following output:

```
a = 0x28ac44, p1 = 0x28ac44, p2 = 0x28ac44.
```

```
a[0] = 10, p1[0] = 10, p2[0] = 10.
a[4] = 50, p1[4] = 50, p2[4] = 50.
*a = 10, *p1 = 10, *p2 = 10.
*(a + 4) = 50, *(p1 + 4) = 50, *(p2 + 4) = 50.
```

The actual three memory addresses you see displayed in the first line will probably not be the same as these, but they will be the same as each other.

The above example shows that arrays can be treated as pointers and pointers can be treated as arrays! This is because, as stated above, the name of an array is a pointer constant to its first element.

Let's look at the code line by line to understand what is really going on here.

The first line of main declares an array of five integers and initializes it. The next two lines declare two pointers to integers.

The statement "p1 = a;" initializes the pointer "p1". Since the name of an array is a pointer constant to the first element of the array, "p1" now also points to the first element of the array. The statement "p2 = &a[0];" explicitly initializes "p2" with the address of the first element of the array. This is identical to the statement "p2 = a;". I'm simply showing two ways to express the same thing and to further demonstrate what the name of an array really indicates.

Of course, each pointer could have been declared and initialized with one line each as follows:

```
int *p1 = a;
int *p2 = &a[0];
```

Or, we could have declared and initialized both pointers with just one line as follows:

```
int *p1 = a, *p2 = &a[0];
```

Remember, when an asterisk (*) is used in a declaration, it simply means that the variable name following it represents a pointer; it does not actually dereference the pointer for the initialization. When used elsewhere, and asterisk is the indirection operator, and the expression "*p1" would refer to the memory that "p1" points to.

Note that since an array name is a pointer "constant", the following line would NOT be valid if "a" is an array and "p" is a pointer:

```
a = p;
```

You can not assign a value to the name of an array. If you try, you will get a compiler error.

We are now going to look at the first "printf" statement.

This line prints out three memory addresses. (Remember, the "%p" conversion-code earlier is used when printing out memory addresses, and it forces the address to be printed out in hexadecimal notation on most systems.) All three addresses specified are the same; they all point to the first element of the array.

Now let's look at the second "printf" statement. All three values printed are the first element of the array (which has value 10). Note that although "p1" and "p2" are declared as pointers, C allows them to be treated as arrays! The next statement is similar; all three values printed are the fifth and final element of the array.

Now look at the fourth "printf" statement. Here, each value printed is the value pointed to by a pointer pointing to the first element of the array! (Again, the name of an array is a pointer constant to the first element of the array.) Of course, these values are all 10. Here, note that although "a" is the name of an array, C allows us to treat it as though it were a pointer.

Now let's look at the last "printf" statement. Remember, the asterisk here is the indirection operator. It takes the value stored at the memory address indicated by the expression to its right. Normally, this expression will be a single pointer, but this is not necessarily the case. Here, we are adding 4 to each pointer to obtain a new memory address. But on most modern systems, each integer requires 4 bytes of storage. So why, then, don't we see the value located at "a[1]"? Because C knows that when programmers are adding and subtracting an integer, let's say "n", from a pointer, they generally want to step "n" units away from the starting point. So, if "p" is a pointer to a particular type, the memory address represented by "p+n" is interpreted as "p+sizeof(type)*n". This is known as **pointer arithmetic**. Given a pointer, p, the expressions p+n and p-n are pointers to values n elements away. In this case, "a+4" is a pointer to the element four slots to the right of the first element in "a", so the value of "a[4]", or 50, is printed. Again, pointers can be treated as arrays and arrays can be treated as pointers. If "x" is either an array or a pointer, and "n" is an integer, then *(x + n) is identical to x[n].

You may understand better now a particular aspect involving functions, arguments, and parameters. Remember that when you pass a normal variable to a function, the function cannot permanently change the value of that variable, but when you pass an array, the function can permanently change the values stored in the array. This is because when you pass an array to a function by specifying its name, you are really passing a pointer constant to the memory associated with the array! The function can write to this memory, thereby changing what is stored in the array, just like a function can change the value that a pointer parameter points to.

It is also possible to subtract one pointer from another of the same type. Rather than just subtracting one memory location from another, C computes the number of elements apart that one pointer is from another. Here is an example that also introduces the concept of casting:

```
#include <stdio.h>

int main(void)
{
  int a[5];
  int *p1 = &a[0], *p2 = &a[4];
  int m1, m2;

  m1 = (int) p1;
  m2 = (int) p2;
  printf("p1 = %p, p2 = %p.\n", p1, p2);
  printf("m1 = %p, m2 = %p.\n", m1, m2);
  printf("p2 - p1 = %d.\n", p2 - p1);
  printf("m2 - m1 = %d.\n", m2 - m1);
  printf("(int) p2 - (int) p1 = %d.\n", (int) p2 - (int)
p1);

  return 0;
}
```

This program might display:

```
p1 = 0x28ac3c, p2 = 0x28ac4c.
m1 = 0x28ac3c, m2 = 0x28ac4c.
p2 - p1 = 4.
m2 - m1 = 16.
(int) p2 - (int) p1 = 16.
```

Look at the two lines that are initializing "m1" and "m2". Normally, we can't assign the value of a pointer to an integer variable. This would cause a compiler error. The "(int)" tells the compiler that we are **casting** the pointer as an integer. The memory address is therefore treated as an integer and assigned to the integer variable.

Of course, when you run this program, you will probably not see the exact memory addresses shown here, but you will see the same values for "m1" and "m2" as you see for "p1" and "p2". The second "printf" is printing the values of the variables "m1" and "m2" in hexadecimal because the "%p" in the format string passed to "printf" is telling "printf" to consider the passed value to be memory addresses.

Now look at the next two "printf" statements. Given that the values of "m1" and "m2" are the same as the values of "p1" and "p2", you might find it quite unusual that the two subtractions are returning two different results. This is because the third "printf" statement is subtracting a pointer from a pointer and the fourth "printf" statement is subtracting an integer from an integer, and these are two different things. When you subtract one pointer from another, the resulting value is the number of elements apart that one pointer is from the other. The last "printf" is explicitly casting the pointers as integers

and subtracting one integer from another, so this returns 16. On some systems, you would see a number other than 16 running this same program, because, again, not all systems use four bytes for each integer, but most today do.

Normally, you can not assign the value of one pointer to a pointer which points to a different type. For example, if "pi" is a pointer to an integer and "pc" is a pointer to a character, the statements "pi = pc;" and "pc = pi;" would each give compiler errors. If you really want to, however, you could avoid the compiler errors by using the statements "pi = (int *) pc;" or "pc = (char *) pi;". These are other examples of casting. In the first case, we are taking a pointer to character and casting it as a pointer to integer. In the second, we are taking a pointer to integer and casting it as a pointer to character. In each case, we are taking the memory address stored in the pointer to the right of the equal sign and assigning it to the pointer to the left of the equal sign. While valid, this type of casting is rarely useful.

However, pointer casting can be useful when we are dealing with void pointers. A void pointer can be declared as follows:

```
void *p;
```

In a sense, we are saying here that "p" is a pointer, but we don't know what it will point to! You can assign any type of pointer to a void pointer, but you can never dereference a void pointer (if you try, you will get a compiler error). For example, if "pVoid" is a pointer to void and "pInt" is a pointer to int, the following statements are valid:

```
pVoid = pInt;
pInt = pVoid;
```

It is probable more readable to cast a void pointer when assigning its value to another pointer as follows:

```
pInt = (int *) pVoid;
```

Some compilers may actually require this, producing an error or a warning if you do not include the cast.

Now we are going to look back at another way to code a program we considered when we first discussed arrays. The program allowed a user to enter 10 integers and printed them out in reverse order. This was the way we did it last time was:

```
#include <stdio.h>

#define ARRAYSIZE 10

int main(void)
{
  int x, numbers[ARRAYSIZE];
```

```
  for (x = 0; x < ARRAYSIZE; x++)
  {
    printf("Enter number: ");
    scanf("%d", &numbers[x]);
  }

  for (x = ARRAYSIZE - 1; x >= 0; x--)
    printf("%d\n", numbers[x]);
}
```

Here is another way to do it:

```
#include <stdio.h>

#define ARRAYSIZE 10

int main(void)
{
  int numbers[ARRAYSIZE];
  int *p;

  for (p = numbers; p < numbers + ARRAYSIZE; p++)
    {

      printf("Enter number: ");
      scanf("%d", p);
    }

  for (p = numbers + ARRAYSIZE - 1; p >= numbers; p--)
    printf("%d\n", *p);

  return 0;
}
```

The logic behind the code is the same; we are simply walking along the array with pointers instead of manipulating indices.

Note that "numbers + SIZE" does not just add 10 to the memory location pointed to by "numbers" (the array name is a pointer constant), but rather adds "10 * sizeof(int)" since we are adding to a pointer. The line "p++" adds "sizeof(int)" to the pointer "p" (i.e., "p" steps right one slot in the array).

Interestingly, this is the first time we see a call to "scanf" without the use of the address operator ('&'). Why? Because "scanf" requires the address of the memory location it will fill in with a value, and that is exactly what "p", the pointer, represents! The "scanf" line here is equivalent to:

```
scanf("%d", &(*p));
```

which would also work but is more complicated than is necessary.

The address "numbers + SIZE - 1" is the address of "numbers[SIZE-1]" which is the address of the last slot of the array. Note that it is also OK to compare one pointer to another, as we do with the comparison "p >= numbers". If "p" is to the right of "numbers" (i.e., it stores a higher memory location), the expression is true. The line "p--" actually subtracts "sizeof(int)" from "p" ("p" steps left one slot in the array).

Neither of the two methods for writing this program is better than the other; it's more a matter of preference. I think most programmers would prefer the first method since you generally don't deal with pointers unless you have to.

Next we are going to look at a similar programming problem that requires pointers!

Let's say we want to write a program that allows the user to enter an arbitrary number of integers and then prints them out in reverse order. Our program will begin by asking the user how many integers will be entered. Then, it will allow the user to enter the integers, and then it will print them out in reverse order.

This task cannot be accomplished with an array. Let's say, for instance, that we expect the user will want to enter less than 10,000 integers, so we declare an array of size 10,000. The user then decided to enter 10,001 integers. Our array is not big enough to store them! Perhaps we will simply not allow the user to type more than 10,000 integers. This might be OK, depending on the application. However, there is still something wasteful going on. Let's say that normally, the user only wants to enter 10 or 20 integers, but we are allocating space for 10,000. If we do things like this often in a large program, it might run out of memory. We want to allocate just enough space to store the integers that will be entered.

Furthermore, there are certain instances in which we want the user to be able to keep entering data as long as there is any memory left. Consider applications such as word processors, databases, or spreadsheets. For this type of application, we can not rely on arrays, since all the memory is allocated once at the beginning of the program. We need to allocate memory during the execution of the program.

Both types of situations just described require **dynamic memory allocation**. The function we are going to discuss which allows dynamic allocation of memory is "malloc". The "malloc" function is provided in the standard library, and to use it, you must include the header file "stdlib.h" at the top of your program. (Like the other header files we have used in the past, this actually contains a list of prototype statements for the functions in the library.) The function "malloc" takes one parameter, which is the number of bytes we wish to allocate, and it returns a pointer to the allocated memory. The prototype statement is for "malloc" is:

```
void *malloc(size_t size);
```

The type "size_t" might differ from system to system. It is also defined in "stdlib.h", and it is usually just an unsigned integer, but it will be something large enough to specify the largest amount of memory that you might allocate.

Note that "malloc" returns a pointer to void, so this pointer can then be assigned to any other pointer. You don't have to cast it, but you should anyway for readability.

If there is not enough memory left to fulfill the request, "malloc" returns a NULL pointer. Good code should always check if the value returned by "malloc" is NULL before proceeding further.

After a program is through using the memory that it has allocated, it is a good idea to free the memory. This is done with the "free" function, also provided in the standard library. The prototype for this function is:

```
void free(void *ptr);
```

This function takes a pointer and frees the memory associated with it so that it can be used again later in the program. When a program terminates, all memory it has allocated is freed automatically anyway, so for small programs, it isn't really essential to free memory while the program is running, but it is important to get in the habit of freeing your own memory anyway.

Here is an implementation of a solution to the problem discussed above:

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

int main(void)
{
  int x;
  int *p1, *p2;

  printf("How many integers will you enter? ");
  scanf("%d", &x);

  p1 = (int *) malloc(sizeof(int) * x);
  if (p1 == NULL)
  {
    printf("NOT ENOUGH MEMORY!\n");
    return 1;
  }

  for (p2 = p1; p2 < p1 + x; p2++)
```

```
  {
    printf("Enter number: ");
    scanf("%d", p2);
  }

  for (p2 = p1 + x - 1; p2 >= p1; p2--)
    printf("%d\n", *p2);

  free(p1);

  return 0;
}
```

The start of this program is something we've seen many times. We are asking the user to enter a number, and using "scanf" to record this integer in the variable "x". This number represents the number of following integers that the user wishes to enter, and we need to allocate enough memory to store them!

Next is the call to "malloc". We pass it one parameter, the number of bytes we wish to allocate. Since "x" is the number of integers we need to store, and each integer requires "sizeof(int)" bytes of memory, "sizeof(int) * x" is the total number of bytes we need allocated. The "malloc" function allocates a contiguous block of memory of this size and returns a pointer to the beginning of it. This pointer will be a pointer to void, so we don't really need to cast it as a pointer to int for the assignment, but it is a good idea to do this anyway for readability, and most programmers do. (Some compilers might even require it.) So, after this line of code, "p1" will point to the first byte of the allocated memory.

Note that following the call to "malloc", we check to see if "p1" equals NULL. If so, this means that "malloc" returned NULL, which means there wasn't enough memory to satisfy our request. If so, we tell the user that there wasn't enough memory and end the program by returning from "main". You can test this by running the program and entering a ridiculously large number.

The first loop is similar to the loop in the previous program. We start "p2" pointing to the same byte as "p1" (the beginning of the allocated memory). We have it walk along as we fill it in one integer at a time, "x" integers in all.

This is probably a good time to note that the memory allocated by "malloc" is not automatically initialized. In this case, it doesn't matter, because we are filling in the entire block of memory with the loop we just discussed, but often, you will want to write code that walks through allocated memory and initializes it with some default value like 0.

The second loop is also similar to the previous program. We start "p2" pointing to the last integer stored in the allocated memory, and we walk along to the left one integer at a time until we have printed them all in reverse (in which case "p2" will reach back to "p1").

After we are done using the allocated memory, we free it. The program remembers how much space was allocated for each block, so if you pass "free" a pointer to the beginning of an allocated block, it will free that entire block. In this specific case, we're not really accomplishing anything be freeing the block, since the program is about to end anyway, and the memory would have been freed regardless. Still, it is a good idea to get into the habit of always freeing memory when you are done using it. If you ever write a large program which often needs to use temporarily allocated memory and you don't free it afterwards, you will be more likely to run out of memory in the middle.

Often after freeing memory associated with a pointer, it is a good idea to set that pointer to NULL. For instance, you will often see code such as:

```
free(p);
p = NULL;
```

Otherwise, the pointer will continue to point to the block that we just freed. Since the running program no longer owns this memory, it is unpredictable what values will be stored in the memory, and if you try to assign new values to the memory the program will likely crash.

Remember, "malloc" and "free" don't only work for integers. We could be allocating memory to store some other data type, like float or char (or user-defined structures, which we have not yet covered). In any case, the argument to "malloc" is just the number of bytes that you need allocated. The function itself is not aware of the type of data being stored. You should never call "malloc" with an argument of 0 or a negative number. The results are unpredictable.

It is also possible to have pointers to pointers, pointers to arrays, arrays of pointers, and even pointers to functions! These will not be discussed now.

Pointers are extremely important in C, and some future topics will depend on them.