Memory and Pointers

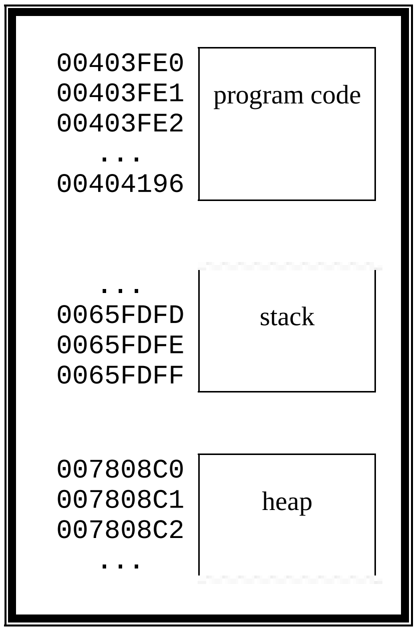
written by Cathy Saxton

Basic Memory Layout

When a program is running, there are three main chunks of memory that it is using:

- A *program code* area where the program itself is loaded.
- A *stack* that is used for storing local variables and maintaining the stack of functions.
- A *heap* that is used when memory is allocated, e.g. using the *new* operator.

Each of these concepts will be explained further below.



The diagram on the right is an example of how memory might look when a program is running. The numbers to the left of each memory block are the addresses (in hexadecimal).

The *program code* area contains all of the functions in your program. When you ask the computer to run your program, it loads the code into memory and begins executing that code.

The *stack* is used to hold local variables. Each time a function is called, its local variables are added to the top of the stack (they are pushed onto the stack). When the function returns, its variables are removed from the stack (they are popped from the stack). This means that the current function will always have its variables on the top of the stack. (In addition, the computer places some additional information on the stack to assist in restoring state when the function returns).

When a program starts execution, the variables for first function (e.g. main()) are pushed onto the stack. As the program runs and calls other functions, they are added to the stack. This means that the stack changes size as the program runs. The diagram to the right shows the stack area with a jagged top line to indicate that the top of the stack changes. The stack is said to grow "up," using memory at lower addresses as it enlarges. Notice that the computer leaves a bunch of memory "above" the stack to allow it room to grow.

The *heap* is used when a program asks to allocate memory, e.g. by using the *new* operator (which is discussed further in the section on allocating memory). The computer will look in the heap for an unused chunk of memory of the requested size. If will return a pointer (reference) to it and will remember that it is in use (so it is not given away again). When the program is done with allocated (heap) memory, it should free it, which makes that heap memory available again. Like the stack, the heap changes size as it is used. In general, the heap grows "down," using higher addresses for each memory allocation, which is shown with the jagged bottom line in the diagram above.

Sizes of Common Variable Types

byte	= 8 bits
word	= 2 bytes = 16 bits
short	= 2 bytes = 16 bits
long	= 4 bytes = 32 bits
int	= 4 bytes = 32 bits (on current operating systems)

An int (integer) is not guaranteed to be a specific size. It will be the size that the system can use most efficiently for mathematical operations. On current computer operating systems, an integer is 4 bytes. On older systems (and many current robotics microcontrollers), an integer is only 2 bytes.

Local Variable Location in Memory

If you have a function with three local variables:

```
int x, y, z;
x = 3;
y = 5
z = 7;
```

the top of the stack might look something like:

Address	Stack	Variable
0065FDEC	7	z
0065FDF0	5	y
0065FDF4	3	x

The stack memory contains the values for each variable. Each variable is located at a specific address. Notice that since we're using integers, each variable requires 4 bytes of storage, so the addresses are 4 bytes apart.

C/C++ Syntax for Memory Access

Pointer Variable Declaration

To declare (create) a variable to use as a pointer to memory, use the following syntax:

```
<type> *<variable>;
```

For example:

```
int *px;
```

creates a variable, *px*, that is a pointer to an integer.

Pointer Variable Initialization

Note that *px* is currently uninitialized; we still need to tell it what to point to. Suppose we want *px* to point to variable *x*; we can ask for the address of *x* using the & operator:

```
px = &x;          // read "px gets the address of x"
```

The above line sets the variable *px* equal to the address (memory location) of *x*. In the above example, the address of *x* is 0065FDF4.

<u>You must set the value of a pointer (make it point to some memory) before you read from it or put something in it</u>.

Setting and Reading the Contents of Pointer Variables

Once a pointer is set to point to an appropriate memory location, we can use it to look at and/or set the contents of that memory. We can set the contents of the memory pointed to by *px* to the value 2 with the following code:

```
*px = 2;          // read "contents of px gets the value 2"
```

We can read the contents of the memory pointed to by *px* and store that value into an integer variable *y* as follows:

```
y = *px;          // read "y gets the contents of px"
```

To input a value from the user and store it into the memory pointed to by *px*, use:

```
cin >> *px;       // or: scanf("%d", px);
```

Output the contents of *px* to the console window with:

```
cout << *px;      // or: printf("%d", *px);
```

Example

The following code shows examples of the declaration and use of a pointer variable *px*. Note that we set the value of *px* (px = &x) before looking at its contents (*px).

```
int main()
{
      int x, y, z;
      int *px;

      x = 5;
      cout << "x = " << x << endl;
      cout << "address of x = " << &x << endl;

      px = &x;
      cout << "px = " << px << endl;
      cout << "contents of px = " << *px << endl;

      y = x;
      cout << "y = " << y << endl;

      z = *px;
      cout << "z = " << z << endl;

      *px = 8;
      cout << "contents of px = " << *px << endl;
      cout << "x = " << x << endl;

      return 0;
}
```

The above program produces the following output:

```
x = 5
address of x = 0x0065FDF4
px = 0x0065FDF4
contents of px = 5
y = 5
z = 5
contents of px = 8
x = 8
```

(Of course, the value for the address of x may be different each time the program is run.)

Exercises

What would you expect the output to be for each of the following?

```
      int y;                           int a, b;
      int *pnum;                       int *pnum;

      y = 7;                           a = 8;
      pnum = &y;                       pnum = &a;
      *pnum = 5;                       b = *pnum;
      cout << "y = " << y << endl;     cout << "b = " << b << endl;
```

Allocating Memory

Above, we used a pointer as a reference to another variable. We set the pointer's value by asking for the address of the other variable.

Another use for a pointer is to point to allocated memory. You can ask for memory from the heap (allocated memory) by using the *new* operator:

```
<pointer variable> = new <variable type>;
```

For example:

```
int *pnum;         // declaration of a pointer to an integer
pnum = new int;    // allocation of an integer; pointer stored in pnum
```

<pointer variable> must be a variable declared as a pointer to <variable type>. In the above example, <variable type> is an integer. To declare a pointer and allocate memory to store a double, use:

```
double *pnum;
pnum = new double;
```

When you are done with allocated memory, use the *delete* operator to free it:

```
delete pnum;
```

Allocating Arrays

Sometimes it doesn't make sense to declare an array as a local (stack) variable (e.g. int anum[7];) when writing code. It may be more convenient to allocate the array after getting information from the user. You can use the *new* operator to allocate an array by using the following syntax:

```
<pointer variable> = new <variable type>[<count of elements>];
```

Suppose that you asked a user how many numbers they wanted to enter and stored that value in a variable *count* (e.g. cin >> count). You could allocate an array of *count* integers as follows:

```
int *list;
list = new int[count];
```

The array *list* would be able to hold as many integers as the user had requested (the value stored in the variable *count*). You can use an allocated array variable just like an array you declared as a local (stack) variable. For example, you might input numbers with a loop like:

```
for (i = 0; i < count; ++i)
      cin >> list[i];
```

To free the memory allocated for an array, put an empty pair of brackets between *delete* and the variable:

```
delete [] list;
```

Allocating Structures

Suppose you had a structure defining the coordinates for a rectangle:

```
struct RECT
{
   int left, top;       // (x, y) for upper left corner
   int right, bottom;   // (x, y) for lower right corner
};
```

Recall that if you declare a local variable for a structure:

```
RECT rect;
```

You can reference its fields by placing a period (.) between the variable and field name:

```
rect.left = 0;
```

You can allocate memory for a structure (instead of declaring a local variable). Here is an example for allocating a RECT struct. The pointer to the memory to hold the struct is stored in the variable *prect*.

```
RECT *prect;
prect = new RECT;
```

When you are referencing fields in a structure pointed to by a variable (e.g. *prect*), you need to use a different syntax. Place an "arrow," a hyphen and greater-than sign (->), between the variable and field name (note that the arrow "points" to the field names):

```
prect->left = 0;
prect->top = 5;
prect->right = 20;
prect->bottom = 35;
```

Passing Pointers in Function Calls

When making a function call, passing by reference is one way to provide the function with a way to modify a variable owned by the caller. Another way to enable a function to change the caller's variable is to pass a pointer to the variable. This is a common practice in Windows programming. Consider a function that asks the user to enter x and y coordinates and wants to return those values to the caller. The caller can pass the addresses of its variables:

```
        int x, y;
        int *px, *py;
        px = &x;
        py = &y;
        GetCoord(px, py);
```

Note that we are passing pointers to integers for both arguments. (Recall that taking the address of a variable resulted in getting a pointer to the variable's memory location). The function *GetCoord* receives pointers to the variables *x* and *y*.

A possible implementation of GetCoord:

```
        void GetCoord(int *px, int *py)
        {
           cin >> *px;
           cin >> *py;
        }
```

Note that the parameters look just like variable definitions for pointers, which matches what we passed to the function.

The above example for code making a call to GetCoord illustrates how the parameters match between the call and the function. It is equivalent to just take the address of the variables when you make the call:

```
        int x, y;
        GetCoord(&x, &y);
```

It isn't necessary to declare and set px and py; you can just pass the addresses directly.

You can also pass pointers to structures. For example, you could call a function to set the values of the above RECT structure:

```
        RECT rect;
        GetRect(&rect);
```

The function *GetRect* might look like:

```
        void GetRect(RECT *prect)
        {
           cin >> prect->left;
           cin >> prect->top;
           cin >> prect->right;
           cin >> prect->bottom;
        }
```

The caller passes the address of a RECT structure (a pointer to a RECT). Since the function has a pointer to the struct, it uses the "arrow" (`->`) syntax to reference the struct's fields.

It is also common to pass a pointer to a structure even when the contents of the structure don't need to change. It is more efficient to pass a pointer (4 bytes) than an entire structure, which can be tens or hundreds of bytes. In this case, the function declares the pointer variable using *const*, which indicates that the contents of the structure can't change. For example:

```
        int CalcArea(const RECT *prect);
```

The *CalcArea* function would be able to look at the fields in the structure:

```
        width = prect->right - prect->left;
```

But, it would not be allowed to modify any fields. The compiler would report an error for code such as:

```
        prect->left = 0;
```

Exercise

Write the following functions (use the given prototypes):

```
void GetRect(RECT *prect);
```

- Prompt the user and input values.
- Make sure the input is legal (e.g. right must be greater than left).
- If the input is not legal, input the coordinates again (until the values are OK).

```
int CalcArea(const RECT *prect);
```

- Calculate and return the area for the given rectangle.

```
void MakeSquare(RECT *prect);
```

- Reset the right side or bottom side to make the *largest* possible square that can be *contained* in the given rectangle.

```
void OutputRect(const RECT *prect);
```

- Output the values for the rectangle in the format "{left, top, right, bottom}".

Write a test program to do the following, calling the above functions as appropriate. In cases where it is outputting information, be sure to also output explanatory text.

- Declare a rectangle as a local or allocate it.
- Ask the user for the four sides of the rectangle.
- Output the area of the rectangle.
- Make the rectangle square.
- Output the resulting (square) rectangle.
- Output the area of the new (square) rectangle.
- (Free any memory that you allocated.)

<u>*Challenge*</u>: Implement and test the function:

```
bool IntersectRect(RECT *prectResult, const RECT *prect1, const RECT *prect2);
```

- If prect1 and prect2 intersect, set *prectResult* to the rectangle resulting from the intersection and return true.
- Otherwise, return false.