Getting the Size of a Variable

The `sizeof` operator can be used to determine the number of bytes of storage occupied by a variable or type of value.

```
#include <stdio.h>

int main()
{
   int x;
   float f;
   printf("'int' variable x takes %d bytes.\n", sizeof(x));
   printf("'float' variable f takes %d bytes.\n", sizeof(f));
   printf("'double' values take %d bytes.\n", sizeof(double));
}
```

`scanf` Revisited

- The `&` is required in a call to `scanf` because it is the *address* of the variable, rather than its *value*, that is being passed.
- The *size* of the variable whose address is passed must *exactly match* the size expected by the corresponding conversion in the format string.
- If the sizes do not match, either the wrong value will be returned or the memory contents will be corrupted.

Examples

Try these on your system:

```
#include <stdio.h>

int main()
{
   double d;
   printf("Type a number: ");
   scanf("%f", &d);
   printf("You typed '%f'\n", d);
}
```

The `double` variable `d` consumes 8 bytes of storage, but `scanf` stores only a *single-precision* (*i.e.* 4-byte) floating point value.

```
#include <stdio.h>

int main()
{
   float f;
   printf("Type a number: ");
   scanf("%lf", &f);
   printf("You typed '%f'\n", f);
}
```

The `l` tells `scanf` to store a *double-precision* (*i.e.* 8-byte) floating point value, but only 4 bytes of storage are allocted for variable `f`.

Initialization of Local Variables

In C, uninitialized local variables contain *unpredictable* values.

```
#include <stdio.h>
void test();

int main()
{
   printf("Hello world!\n");
   test();
}
void test() {
   int i, j, k;
   printf("i = %d, j = %d, k = %d\n", i, j, k);
}
```

The programmer *must* remember to initialize local variables before use!

Pointer Variables and Values

- The declaration

  ```
  double d;
  ```

  defines `d` to be a variable of type `double`. The *values* that can be assigned to `d` are double-precision *floating point numbers*.

- The declaration

  ```
  double *dp;
  ```

  defines `dp` to be a variable of type "pointer to `double`." The *values* that can be assigned to `dp` are *addresses*, not floating point numbers.

In a *declaration*, the operator `*` means *pointer to*.

Dereferencing and Assignment

In C *expression*, the `*` operator is used to *dereference* (*i.e.* follow) a pointer to obtain the pointed-at value.

- `d = *dp` means:

  "Get the `double` value *whose address is stored in* `dp`, and write that value in the variable `d`."

- `*dp = d` means:

  "Get the value of `d` and write that value in the variable *whose address is stored in* `dp`."

- `dp = d` and `d = dp` are *illegal* (type mismatch).

In an *expression*, the operator `*` means *dereference*.

The Pointer Value `NULL`

The value `NULL` is defined in the system header file `stdlib.h`:

```
#define NULL ((void *)0)
```

The value `NULL` serves essentially the same purpose in C as the null object reference `null` does in Java.

Attempts to dereference `NULL` will (on most modern general-purpose systems) cause your program to “dump core”.

L-values and R-values

- An *L-value* is an *address*.
- An *R-value* can be any type of value.
- On the *left-hand side* of an assignment, an expression is evaluated to obtain its *L-value* (the address of the variable to be changed).
- On the *right-hand side* of an assignment an expression is evaluated to obtain its *R-value* (the value to be stored).

- Only certain kinds of expressions have L-values:

```
int x, *y;

   x = 3;
   y = &x;
  *y = 0;
y[x] = 5;
```

but *not*

```
x+3 = 4;
```

Only expressions that have *L-values* are permitted on the *left-hand side* of an assignment.

The & and * Operators

- The `&` operator is used to obtain the *address* of a variable:

```
int x, *y;
y = &x;
```

- The `*` operator is used to *dereference* an address to *access the pointed-at variable*:

```
x = *y;
*y = 0;
```

Uninitialized Pointer Variables

The declaration:

```
int *ip;
```

allocates a variable `ip`, but it *does not assign it any initial value*.

- In the absence of initialization, the code

  ```
  printf("%d\n", *ip);
  ```

  will produce either *garbage output* or *crash*.

- Or, the code

  ```
  *ip = 3;
  ```

  will *corrupt the contents of memory*, perhaps crashing immediately, perhaps crashing later, or perhaps producing incorrect results or strange program behavior.

Initialization of Pointer Variables

Pointer variables must be initialized before they are dereferenced:

```
int i, *ip;
ip = &i;
*ip = 3;
```

Here, the variable `ip` is initialized to point to `i`, and then `3` is assigned to `i` (by dereferencing the pointer `ip`).

Usually, pointers are initialized to point to *anonymous storage* that is allocated using the C library function `malloc()`.

A Pointer Quiz

Suppose the following declarations have been executed:

```
int i = 3, j = 5, *p = &i, *q = &j, *r;
double x;
```

What will be the values of the following expressions?

1. `p == &i`
2. `**&p`
3. `r = &x`
4. `*(r = &j) *= *p`
5. `7**p/*q+7`