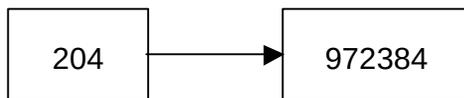# Pointers and Strings

Basics

- *A pointer variable contains a memory address as its value*. Normally, a variable contains a specific value; a pointer, in contrast, contains the memory address of another variable. We usually depict a pointer variable p as a container with an arrow to the variable whose address it contains; in the picture below, p contains the memory address 204, which contains the number 972384.


```
int x = 972384;
int *p; // p can point to an int
p = &x; // p gets address of x
```

Memory contents

Declaring Pointers

To declare that a variable named p can contain the address of an object of type T, use the notation
```
T *p;
```

For example:
```
int *ptr_to_int;
double *ptr_to_double;
string *ptr_to_string;
```

To declare that two or more variables can all point to objects of type T:
```
T *p1, *p2, *p3;
```

not
```
T *p1, p2, p3;  // only p1 is a pointer here
```

Assigning To Pointers

The value assigned to a pointer must be an address. C (and therefore C++) has an address-of operator & whose value is the address of its operand.

```
int x = 972384;
int * p;
p = &x;      // &x is the address-of x
```

Although it is the same symbol as the reference operator, it is easy to know which is which if you remember the following rule:

If the symbol "&" appears in a declaration, it is declaring that the variable to its right is a reference variable, whereas if it appears in an expression that is not a declaration, it is the address-of operator and its value is the address of the variable to its right.

```
int x = 972384;
int y = 100;
int &ref = x;  // a declaration, so & is the reference operator
int  *p = &x;  // appears on right of =, so it is in an
               // expression and is address-of operator
p = &y;        // address-of operator
int &z = y;    // reference operator
```

Dereferencing Pointers

Accessing the variable to which a pointer points is called ***dereferencing the pointer***. The "*" operator, when it appears in an expression, dereferences the pointer to its right. Once the pointer is dereferenced, it can be used as an l-value or an r-value just like a normal variable. Continuing the program from above,

```
cout << *p;   // p contains address of y, which stores 100,
              // so 100 is printed. This is an r-value
cin >> *p;    // store input into variable pointed to by p,
              // which is y, so input it stored in y
              // this is an l-value
*p = *p + *p; // just like y = y + y
*p = *p * *p; // just like y = y * y
```

The dereference operator and the address-of operator have higher precedence than any arithmetic operators, except the increment and decrement operators, ++ and --.

Note that if you forget the dereference operator when accessing a pointer you will get unexpected results.

```
cout << p;
```

will display the memory address stored in p, not the value stored in y. If you want to print addresses, you can use either of these methods:

```
cout << "Address of y is " << &y;
cout << "Address stored in p is " << p;
```

Const Pointers and Pointer to Constant Data

Sometimes you need a pointer that will point to data that is not allowed to be changed. This is called a pointer to constant data. To declare that a pointer is allowed to point only to data that cannot be changed, use the syntax:

```
const int *ptr;  // data cannot be changed through ptr
ptr = &x;
*ptr += 1;    // syntax error
x = x + 1;    // perfectly legal though
```

In contrast, sometimes you may want a pointer that can only point to a fixed memory location, but the contents of that location can be changed. In other words, the address stored in the pointer is constant, but the contents of that address are not. In this case the pointer must be assigned an address when it is declared. The syntax is

```
int * const ptr = &x;  // ptr's contents cannot be changed
*ptr += 1;    // legal -- adds 1 to x
ptr = &y ;    // syntax error -- tried to change pointer
```

Relationship Between Pointers And Arrays

An array name can be treated like a constant pointer; i.e., it can be used like a const pointer to its base elements, and it contains the address of its first element.

```
int list[5] = {2,4,6,8,10};
cout << *list << endl;  // prints 2
cout << *(list + 2) << endl; // prints list[2], which is 6

int * const aPtr = list;
// aPtr[0] is the first element of list
// &list[0] is stored in aPtr.
aPtr[2] += 10; // changes list[2]
```

The fact that array names are essentially constant pointers is the key to being able to create arrays of arbitrary size while the program is running, rather than by declaring them statically (at compile time).

Strings as Char Pointers

It should now be a little less mysterious why strings can be created using the notation "char *". A C string is just a pointer to a place in memory where a sequence of characters begins, and is assumed to end with a NULL character. The pointer can be used like an array variable and subscripted, as in string[6], because pointers and array names are almost the same.