

Pointer Declaration and Usage

When a pointer is declared, it may be uninitialized or initialized as part of the declaration.

```
char *ptr;           // uninitialized, holds junk
void *x = NULL;     // initialized to "null pointer", remember NULL is #define'd to be 0
int i = 10;
int *p = &i;        // pointer initialized with the address of i using the "address of" operator &
```

```
*p = 0;             // changes the value of i to 0, since p has the address of i, and *p is the location of i
p = NULL;          // changes p to the address 0, no longer points at i
```

Pointers are used quite often in C++ since objects are often allocated from the run-time heap. Heap allocated objects require that you use an appropriately typed pointer to reference an object allocated from the heap using the C++ operator "new".

```
int
main()
{
    int x;           // x is automatically allocated on the run-time stack (4 bytes)

    int *p = new int[10]; // p is automatically allocated on run-time stack (4 bytes), but the
                        // array of 10 integers is explicitly allocated from the run-time heap
    ...
    delete p;        // heap allocated objects must be explicitly deallocated using delete on pointer
    // x is deallocated when scope of main is exited
}
```

8/5/00

1 of 8

Summary of Data Storage Areas

A program typically has three conceptual run-time read/write data areas (code area is read-only): **global memory**, **stack memory**, and **heap memory**. The compiler arranges for program data to be allocated from one of these three areas depending on how the data is declared in the program.

Typically, a data declaration reserves `sizeof(type)` bytes of storage for the data at run-time. Data storage for global data, static file scope data, and string literals are allocated, and initialized, from global memory before "main" is called. This storage is reserved for the entire execution of the program.

All other data are declared within some block scope. The compiler arranges to automatically allocated stack space for the data when the block is entered at run-time if the data is declared non-static. The stack space is deallocated when the scope is exited (for example, a procedure returns to its caller).

Data declared static in a block is allocated the first time the block is entered, and "lives" until the program terminates execution.

At program start, an initial amount of heap space is allocated to the program, and explicit heap allocations are made at run-time by calling the C++ operator "new", which returns the address of a chunk of memory *at least* as big as the size of the requested object. Heap memory can grow up to the maximum virtual address space size allowed by the operating system, but rarely does in a correctly running program. All calls to "new" should eventually lead to a call to "delete" to return storage chunks to the heap so that they can be reused.

8/5/00

3 of 8

Scope of a Data Declaration

The scope of a declaration is a key concept in a language like C++. It defines both the *visibility* and *lifetime* of a variable.

Variables or functions declared at *file scope* either have global or local visibility, depending on storage class, and a lifetime of the entire execution time of the program.

```
// Sample.C file
int some_global_variable = 1; // visible in other .C files
static const char* version = "Version 1.0"; // visible only in this file
```

The { and } tokens define a *block scope* in C/C++. Variables declared within a block, only have visibility within that block. This includes any variables expressed as formal arguments to the procedure/function that names a block.

```
int
some_procedure (int x, int y) // x and y visible in the procedure block only
{
    static int buffer[10]; // a static allocation within a block survives the block
    for (int i = 0; i < 10; i++) {
        int x = buffer[i]; // x is local to the for block, hides argument x
        ...
    }
    // i is still visible at this point
} // i no longer "live" when scope is exited ---- WHY?
// buffer is still in existence for the next time the block is entered
```

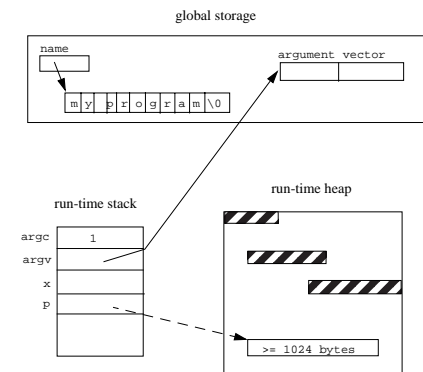
8/5/00

2 of 8

Data Declarations and Storage Allocation

```
char *name = "my program";

int
main(int argc, char* argv[])
{
    int x;
    char* p;
    ...
    // array of 1024 bytes
    p = new char[1024];
    ..
    // delete array
    delete p;
}
```



8/5/00

4 of 8

Stack versus Heap Allocated Data

Pointers can be used to point at global, stack, or heap allocated data, BUT you can only explicitly free heap allocated data. Trying to explicitly delete a global or stack allocated data item will most likely corrupt your running program.

```
static char buffer[1024];

int
some_function(int x)
{
    int *p = &x;    // p points at the address of argument x on the stack

    ...
    delete p;      // Compiler says OK, because p a pointer, but results in run-time error since
                  // the heap doesn't "own" stack memory addresses.

    ...
    delete buffer; // run-time ERROR, buffer allocated in global storage
    ...
}
```

The simple rule is that if you allocate data using operator new, then you must eventually free it using operator delete; otherwise, you will have what is called a “memory leak”, which could eventually lead to memory exhaustion in your running program---meaning you eventually use up all of the virtual memory the operating system allocates to the process running your program.

New and Delete

In C++, operator new is typically defined to call the C library function “malloc” (heap memory allocator), and operator delete calls the C library function “free” to tell the heap allocator that the chunk obtained from a previous call to malloc is no longer needed. If heap memory is exhausted, malloc will return NULL, and new will abort the program unless you define an special “handler” function to cope with the “out of memory” error condition.

```
void*
operator new (unsigned long size)
{
    void *p = malloc (size);    // call C library memory allocator
    if (p == NULL)
        return new_handler (); // call special handler, may abort program
    else
        return p;
}

void
operator delete (void* p)
{
    if (p != NULL)
        free (p);              // call C library free routine
}
```

The compiler takes care of computing the size passed to operator new, and *type casting* the void* (opaque) pointer returned from new to an appropriately typed pointer T*. For example:

```
new T; really means (T*) (operator new (sizeof(T)));
```

```
new T[SIZE]; really means (T*) (operator new (sizeof(T) * SIZE));
```

Stack versus Heap Allocated Data

You have to be careful not to return a pointer to a scope outside of the scope in which a stack allocated data is defined:

```
char*
some_function()
{
    char buffer[SIZE];
    ...
    return buffer; // what happens to buffer when scope is exited?
}
```

What about the following?

```
char*
some_function()
{
    static char buffer[SIZE];
    ...
    return buffer; // what happens to buffer when scope is exited?
}
```

What about a heap allocated buffer?

```
char*
some_function()
{
    char buffer = new char[SIZE];
    ...
    return buffer; // what happens to buffer when scope is exited?
}
```

Reference Variables

A reference variable is similar to a pointer, but has different semantics. The way to think of them is as an alias for a location rather than a pointer to a location, and so they must be initialized when they are declared.

```
int foo = 0;
int& bar = foo; // bar references the same location as foo.
bar = 10;       // changes the value of foo to 10
```

References are most commonly used in procedure argument lists to effect *call-by-reference* parameter passing.

What happens in each of the following cases?

```
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

void swap(int& a, int& b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```