

FYI

1. UT Co-op says that 12 books are on order and should be here this week. There are 2 used books available now.

2. I have put the first homework problem set on my web page:

<http://www.cs.utexas.edu/users/lavender/EE360C>

3. I have updated my web page with additional course information, such as how to submit programming assignments. You should familiarize yourself with this information. You should also check the web page frequently for new and updated information.

4. The GNU Make and GNU Debugger (gdb) postscript manuals are on the web page if you want to read them.

5. I am now holding office hours in ENS 108 from 12:30-1:00 MTWTh.

6. The course Usenet newsgroup is utexas.class.ec360c, but does not appear to be working correctly yet. You can try to access this from the course web page using Netscape from a campus machine, assuming you have a news server specified in your preferences (e.g., news.cc.utexas.edu). Alternatively, you can use one of the news readers available on most Unix machines.

Clarification on Pointer Arithmetic

What happens in the following cases?

```
char buffer[1024]; // 1*1024 bytes of storage
char* bp = buffer;
char* ep = buffer + 1024;

int size = ep - bp; // what value is size?

int a[10]; // 4*10 bytes of storage
int *start = &a[0];
int *end = &a[10];

int size = end - start; // what value is size?
```

What about the following case?

```
void *p = &a[0];
void *q = &a[10];

int size = q - p; // what value is size?
```

What about pointer comparison?

```
if (ptr1 == ptr2) // tests the equality of two integer addresses
    ...

if (*ptr1 == *ptr2) // tests the equality of the values of the two objects pointed at
    ...

if (&ptr1 == &ptr2) // tests the equality of the addresses of two pointers
    ...
```

A Bit More on Variable Declarations

C++ differs from C in that you are **not required** to declare all types at the start of a block, you just need to declare them before they are used. It's really a matter of programming style which you choose to do. I tend to declare pointers at the place in the code where they are initialized. This works because the number of lines in each function I write tends to be small, so it's easy to find the declaration.

```
int C_function()
{
    int x = 0;
    char *p; // declare all local variables at the start of the block
    ...
    p = (char*) malloc(n);
    ...
}
```

In C++, you can declare and initialize a variable at the point where it will first be used.

```
int Cpp_function()
{
    int x = 0;
    ...
    char* p = new char[n]; // declare variable the first time it is used.
    ...
}
```

Structures (A Prelude to Classes)

Structures are used to define simple aggregate types, which are an ordered collection of other types. Each data type of a structure defines one or more elements of members of the structure. In C++, a structure is just a simple type of class---one whose members are all public, meaning visible outside the scope of the {} defining the struct.

```
struct Pair {
    int x, y;
};
```

What is the value of `sizeof (Pair)`?

In C++, a struct defines a new type, which can then be used to declare variables of that type. Since the data elements of a struct are publicly visible, they can be accessed directly. A struct member is accessed using the C++ **member access operator** `'.'`

```
Pair p;
p.x = 0;
p.y = 1;
```

Alternatively, you can initialize a struct with a struct initializer, but only when it is first declared.

```
Pair p = { 0, 1};
```

More frequently, a structure object is allocated from the heap and is referenced by a pointer of the type of the structure. Each member element is accessed using the C++ **member access operator** `'->'`

```
Pair *p = new Pair;
p->x = 0; // same as (*p).x = 0
p->y = 1; // same as (*p).y = 1;
```

Structures (A Prelude to Classes)

Structures quite often contain pointers to other types of objects, including objects of the same type:

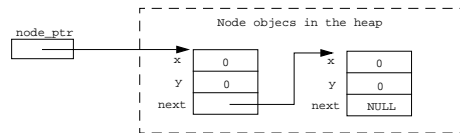
```
struct Node {
    int x, y;
    Node* next; // C programmers would write struct Node*
};
```

What is the value of `sizeof(Node)`?

```
Node *node_ptr = new Node;
node_ptr->x = 0;
node_ptr->y = 0;
node_ptr->next = NULL;
```

Later we might allocate another node, and connect it to the first one, so that we end up with a *linked list* of Node objects.

```
node_ptr->next = new Node;
node_ptr->next->x = 0;
node_ptr->next->y = 0;
node_ptr->next->next = NULL;
```



Structures and Pointers

Initializing structure fields manually is rather tedious, so we would prefer to write a function to create a new Node object and return a pointer to the new Node, and a corresponding function to delete a node.

In C++, we can define a **default value** for an argument to a function, which is used unless the caller explicitly provides a value for the argument.

```
Node* new_node(int x, int y, Node* next = NULL)
{
    Node* node = new Node;
    node->x = x;
    node->y = y;
    node->next = next; // may or may not be initialized to NULL
    return node;
}

Node* delete_node(Node* node)
{
    Node* next = node->next;
    delete node;
    return next; // return next node in the list, may be NULL
}

Node* root_node = new_node(0,0);
Node* node = root_node;
for (int i = 1; i < 10; i++) {
    node->next = new_node(i, 0);
    node = node->next;
}
```

To delete all the nodes is trivial using a for loop that “walks” the list deleting each node.

```
for (Node* node = root_node; node != NULL; node = delete_node(node))
    ; // semicolon alone is an empty statement
```

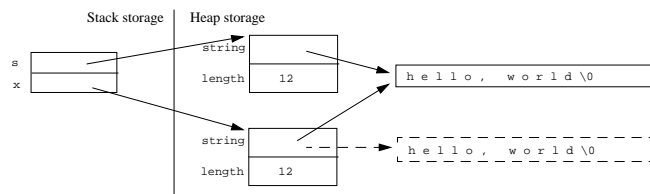
Structure Copying

A type compatible structure may be assigned to another, which results in a shallow copy being made.

```
struct String {
    char* string;
    int length;
};

String *s = new String;
s->string = new char[15];
strcpy(s->string, "hello, world"); // C library function to copy a '\0' terminated string
s->length = strlen(s->string); // C library function to compute string length
```

```
String x = s; // shallow bitwise copy of s, which may not be what we want!
```



Structure Copying

It is often better to write a structure copy function that will do a “deep” copy.

```
String* string_copy(const String& from) // why a const String& argument?
{
    String *to = new String;

    to->length = from.length;
    to->string = new char[to->length + 1]; // length + 1 extra for the terminating '\0'

    if (from.string != NULL) // strcpy will abort if from.string is NULL
        strcpy(to->string, from.string);
    else
        *to->string = '\0'; // null string, which is different than a NULL pointer

    return to; // caller has to remember to delete the string object
}
```

Use the string copy function like so:

```
String s = { "hello, world", 12 };
String *t = string_copy(s);
```

Since `string_copy` requires pass-by-reference, if we have a `String*`, we have to dereference it first:

```
String *s = new String;
... // initialize members of s;
String *x = string_copy(*s); // dereferencing a String* yields a String, which can be passed to String&
```

NOTE: If you attempt to dereference a NULL pointer, your program will abort with a “segmentation violation” (SIGSEGV - signal 11 on Unix) as address 0 is a non-readable address location.

Procedure/Function Declarations

In C/C++, there is no real distinction between a procedure and a function. The terms are used interchangeably. Strictly speaking, a function takes arguments passed by value and returns a value without any side-effect; whereas a procedure may generate a side-effects---for example by using a pointer to effect a modification on some object defined outside of the procedure. In C++, a procedure is commonly called a function.

All function declarations (“prototypes”) specify the **return type**, the **name of the function**, and the **type and number of arguments** passed to the function. If a function does not return a value, then its return type is **void**. Similarly, if a function takes no arguments, you can define the argument list as **void**:

```
extern void swap (int& x, int& y); // a C++ function to swap two integer values
extern void abort(void); // (void) is the same as () --- meaning no arguments
```

If you want to call a C function from C++, the compiler must be shown a C function prototype, or it won't generate the correct linkage information (i.e., the C function name with a leading '_', such as _malloc):

```
extern "C" void* malloc(unsigned long);
extern "C" void free (void*);
```

You can specify external linkage to several C function prototypes in an extern "C" block:

```
extern "C" {
char* strcpy(char* to, const char* from);
int strlen(const char*); // Note that only a type is required, not a variable name
}
```

Usually, you don't have to do this unless you write the C functions yourself. All the standard C library functions are defined this way in the standard system header files. For example, look in /usr/include on most Unix systems (e.g., /usr/include/stdio.h and /usr/include/string.h).

Printf Format Conversion Specifiers

printf defines several conversion specifiers, the most useful ones are:

```
%d converts an integer value to a decimal string
%u converts an unsigned integer value to a decimal string
%x converts an integer value to a hexadecimal string
%f converts a float or double value as a string [-]ddd.dddd
%e converts a float or double value to a string [-]d.ddde-dd
%c prints a char as a one character string, Null characters are ignored
%s prints characters starting at a char* until a '\0' character is found
%p prints a pointer in an implementation dependent manner (usually hex)
```

In C++ programs, we will generally avoid use of the C standard I/O library functions like printf. Instead, we will use the C++ ostream library, which defines an object-oriented set of stream objects and functions for doing type-safe formatted input and output.

Briefly, for formatted output in C++, there is a special output stream object **cout**, that has a special operator **<<** defined for sending output into the stream to be formatted and printed:

```
int x = 5;
cout << x << '\n';
```

For formatted output, you can just direct each part of the output into the output stream.

```
int x = 5;
int y = 10;
cout << "x = " << x << " times y = " << y << " is equal to " << x*y << endl;
```

Procedure/Function Declarations

C++ defines a special function argument type called “ellipsis”, which means an unspecified number of arguments. For example, the C function printf, which is part of the C standard I/O library (stdio), would be defined in a C++ function prototype as follows:

```
extern "C" void printf (const char* format, ...);
```

The first argument is a constant character string, followed by an unspecified number of additional arguments that are NOT type checked. To understand the ellipsis, you have to know a bit about how printf is used for formatted output:

```
int x = 5;
int y = 10;
printf("x = %d times y = %d is equal to %d\n", x, y, x*y);
```

The first argument to printf is a literal string, called the format string, that uses the special conversion specifiers prefixed by the '%' symbol to indicate the type of data object that will be printed. In this case, a string and a decimal integer value. So, in this case, there are 3 %d conversion specifiers, so printf will expect to find 3 integer arguments on its run-time stack.

In the following case, printf expects to find the starting address of a character string that is terminated by the null character '\0' used to terminate a string literal.

```
char* str = "hello, world"; // string literals are terminated by a '\0' character
printf("%s\n", str);
```

Formatted printing to a memory buffer

The C function sprintf, is similar to printf, but is used to “print” a formatted string to a buffer in memory.

```
extern "C" int sprintf(char* buf, const char* format, ...);
```

Sprintf is sometime useful when you want to create a formatted string, and then print it later using a C++ output stream object like cout. The only drawback to this approach, is that you have to guess how big to make the print buffer. Usually, programmers opt to waste space in order to be safe and not overflow the buffer.

```
char buffer[1024]; // big enough to hold the formatted string.
sprintf(buffer, "The value of x is %d", x);
...
cout << buffer << endl;
```

We will see later that C++ defines a special stream object, called a streambuf, that grows dynamically based on the amount of data that is put into the streambuf. For now, a C++ input stream object and a fixed size char array can be used together to read strings from the standard input:

```
int main()
{
    char login[128];
    char password[128];

    cout << "login: ";
    cin >> login;
    cout >> "password: ";
    cin >> password;

    cout << "SunOS 5.5 ... " << flush; // you can flush the stream
    ...
    return 0;
}
```