Pointers and constants:

cchar buer[100];
char name[20];
const char *p = buer;
   // pointer to const char
p = name;  // ok
*p = 'x';  // error!
char * const p = buffer;
   // const pointer to char
*p = 'x';  // ok
p = name;  // error!
const char space = ' ';
const char *p = &space;  // ok
char *q = &space;  // error!

Given the following:

char c;
const char cc = 'a';
char *pc;
const char *pcc;
char *const cpc = &c;
const char *const cpcc = &cc;
char *const *pcpc;

which of the following are legal and which illegal?

c = cc;
cc = c;
pcc = &c;
pcc = &cc;
pc = &c;
p = &cc;
p = pcc;
p = cpcc;
*pc = *pcc;
*pc = *pcpc;
*pc = **pcpc;

int a[10];
is equivalent to
int* const a = &a[0];
plus (the compiler) allocates 10 words.

int b[10];
b = a;  // copy array a into b.
   // Won’t work, and is illegal.

What if b had been declared as
int* b;

A string is an array of characters terminated
by '\0':
char s[5] = “ABCD”;
cout << s;
char* t;
t = s;
cout << t;
Arrays
Arrays can be initialized:

```c
int x[3] = { 5, 6, 7};
x[3] = {3, 4, 5}; // but not assigned like this!
```

Pointers and arrays
Name of an array is a const pointer to first el:

```c
int v[4] = { 1, 2, 3, 4};
int* p1 = v; // == int* p1 = &v[0];
int* p2 = &v[0];
(*v)+1 = 10; // but check this one to be sure.
```

```c
void f(char v[])
{
    for(int i=0; v[i] != 0; i++) use(v[i]);
}
```

is equivalent to:

```c
void f(char v[])
{
    for(char* p=v; *p != 0; p++) use(*p);
}
```

The result of “++” etc. depends on the type of object pointed to. The compiler takes care of this.

Pointers and Constants

“Prefixing” a declaration either const makes the object, not the pointer, a const. To declare the pointer to be constant, use *const.

```c
void f(char* p) {
    char s[] = "abcd";
    const char* pc = s; // ptr to const
    pc[3] = 'e'; // illegal
    pc = p; //okay

    char *const cp = s; // constant pointer
    cp[3] = 'a'; // ok
    cp = p; // illegal

    const char *const cpc = s; // const ptr to const
    cpc[3] = 'a'; // illegal
    cpc = p; // illegal
}
```

Constants

```c
const double pi = 3.14;
const int v[] = {1, 2, 3, 4};
const int x; // illegal; need to initialize
```

```c
void g(const X* p)
{
    ... can’t modify *p ... 
}
```

```c
void h() {
    X val; // val can be modified
    g(&val); // g() still can’t modify val
}
```

consts should always be used in place of “magic numbers”.

Pointers and Constants (contd)

No const*, so const before * applies to the preceding type.

```c
char *const cp; // const ptr to char
char const* pc; // ptr to const char
const char* pc2; // ptr to const char
```

Read left to right.
A reference is an alternative name for an object. Mainly used for arguments/return values.

```cpp
void g() {
    int ii = 0;
    int& rr = ii; // must be initialized
    rr++; // ii is incremented.
    int* pp = &rr; // points to ii;
}
```

A reference is not an object; cannot have a pointer to a reference.

```cpp
void increment(int& aa) {
    aa++;
}
```

```cpp
void f() {
    int x = 1; increment(x);
    cout << x; // prints 2
}
```

Pointers, ... (Section 6.2)

The table in at the start of 6.2 lists all the operators in C++; browse through it often!

"lvalue" means "left value", i.e., the sorts of things that can appear on the left of an assignment.

A strange example:

```cpp
void f(int x, int y) {
    int j=x=y; //value of "x=y" is final val. of x
    int* p = &++x; // p points to x;
    int* q = &(x++); // error: x++ is not an lvalue
    int* pp = &(x>y?x:y); // address of x or y
}
```

Unusual syntax:

```cpp
f1(x, y); // call f1 with x, y as arguments
f1((x, y)); // NOT the same thing!
    // y is the only arg.
```

Better: Use `char* strcpy(char*, const char*)`; //from <string.h>

Pointers to Void

A pointer to any type of object can be assigned to a variable of type void*.

```cpp
void f(int* pi) {
    int* pv = pi; // ok
    pv; pv++; // illegal; can't increment void*
    int* pi2 = static_cast<int*>(pv);
    double* pd1 = pv; // illegal;
    double* pd2 = pi; // illegal;
    double* pd2 = static_cast<double*>(pv);//unsafe
}
```

Reading: Chapter 5
Pointers, . . . (Section 6.2)

E.g.: Simple arith. expressions over integers:

struct Exp {
  int kind; // 1: simple; 2: sum; 3: prod
  int val; // if kind is 1
  Exp* left; // if kind is 2 or 3
  Exp* right; // if kind is 2 or 3
};

int Val(Exp* e) {
  if (1 == e->kind) return(e->val);
  if (1 == e->kind)
    return(Val(e->left) + Val(e->right));
  if (2 == e->kind)
    return(Val(e->left) * Val(e->right));
  return(-1);
}

Heap (or “Free store”), . . . (Section 6.2)

How do we create Exp objects?

Two ways:

Exp e1; // allocated on stack
e1.kind = 1; e1.val = 20;
e1.left = 0; e1.right = 0;

Exp* sumExp( const Exp* ep1, const Exp* ep2) {
  Exp* ep = new Exp; // allocated on heap
  ep->kind = 2; ep->left = ep1;
  ep->right = ep2; return ep; }

Exp* prodExp( const Exp* ep1, const Exp* ep2) {
  Exp* ep = new Exp; // allocated on heap
  ep->kind = 3; ep->left = ep1;
  ep->right = ep2; return ep; }

Exp e1; // allocated on stack
e1.kind = 1; e1.val = 20;
e1.left = 0; e1.right = 0;

Exp e2; // allocated on stack
e2.kind = 1; e2.val = 30;
e2.left = 0; e2.right = 0;

Exp* ep1 = sumExp(e1, e2); // on heap
Exp* ep2 = prodExp(e1, e2); // on heap
Exp* ep3 = sumExp(ep2, ep2); // on heap
Exp* ep4 = prodExp(ep1, ep1); // on heap

cout<<"Value of ep4";
printExp(ep4);

Heap (contd.)

printExp void printExp( const Exp* ep) {
  if (1==ep->kind) cout<<ep->val;
  if (2==ep->kind) {
    printExp(ep->ep1);
    cout<<' ' + ' ';
    printExp(ep->ep2); }
}

But: Need to “release” the space from the heap, else “memory leaks”.

delete ep1;
delete ep2; // etc.

Summary: Stack objects are handled by the system; heap objects must be allocated by calling new and released by calling delete.

23

Heap (contd.)