Compiling and Running a C Program in Unix

Simple scenario in which your program is in a single file: Suppose you want to name your program `test`.

1. edit your source code in a file called `test.c`
2. compile your source code: `gcc -o test test.c` The ”`-o test`” part of the command creates an executable file called `test`. (The ”o” stands for ”output”.) If you left off the ”`-o test`”, the executable would be in a file called `a.out`.
3. if compile time errors are discovered, then go to step 1
4. run your program simply by typing `test` (or `a.out`)
5. if run time errors are discovered, then go to step 1

Structure of a C Program

A C program is a list of **function** definitions. A function is the C equivalent of a method in Java.

Every C program must contain one, and only one, function called `main`, where execution begins.

Functions are *free-standing* (not contained inside any other entity).

```
#include <stdio.h>
int main() {   /* this is a comment */
  printf("Hello, world.\n");
}
```

- The `#include` tells the compiler to include the *header* file `stdio.h`, which contains various declarations relating to standard input-output.
- For now, ignore the return value of `main`.
- `printf` is the function that prints to the screen.
- The `\n` is the newline character.
- Comments are delimited with `/* ... */`. *Remember to end your comment!*

A Useful Library

See the Reek book (especially Chapter 16) for a description of what you can do with built-in libraries. In addition to `stdio.h`,

- `stdlib.h` lets you use functions for, e.g.,
  - arithmetic
  - random numbers
  - ways to terminate execution
  - sorting and searching
- `math.h` provides more advanced math functions (e.g., trigonometry)
- `string.h` has string manipulation functions

Printf

The function `printf` is used to print the standard output (screen):

- It can take a *variable* number of arguments.
- The first argument must be a string.
- The first argument might have embedded in it some “holes” that indicate they are to be filled with data.
- A hole is indicated by a percent sign (`%`) followed by a *code*, which indicates what type of data is to be written.
- Following the first argument is a series of additional arguments that supply the data for the holes.

Example:

```
printf("Name = %s, age = %i, fraction = %f",
               "Fred",   30,            .333 );
```

Output is:

```
Name = Fred, age = 30, fraction = 0.333000
```

Variables and Arithmetic Expressions

The main numeric data types that we will use are:

- `char`
- `int`
- `double`

Variables are declared and manipulated in arithmetic expressions pretty much as in Java. For instance,

```
#include <stdio.h>
int main() {
  int a = 3;
  int b = 4;
  int c = a*b - b/3;
  printf("answer is %i\n", c);
}
```

However, in C, all variable declarations of a block must precede all statements.

Reading from the Keyboard

The function `scanf` reads in data from the keyboard.

```
int num1, num2;
scanf("%i %i", &num1, &num2);
```

- `scanf` takes a variable number of arguments
- The first argument is a string that consists of a series of “holes”.
- Each hole is indicated by a percent sign (%) followed by a code indicating type of data to be read.
- After the first argument is a series of arguments, corresponding to the holes in the first argument.
- *The subsequent arguments must each be preceded by an ampersand!* (Related to pointers.)
- The code for an `int` is `%i`; the code for a `double` is `%lf` (not `%d`, as for `printf`).

When you run this program, it will wait for you to enter two integers, and then continue. The integers can be on the same line separated by a space, or on two lines.

Functions

Functions in C are pretty much like methods in Java (dealing only with primitive types). Example:

```
#include < stdio.h >
double times2 (double x) {
  x = 2*x;
  return x;
}
main () {
  double y = 301.4;
  printf("Original value is %f; final value is %f.\n",
          y, times2(y));
}
```

- Functions must be declared before they are invoked. Requires careful ordering or use of prototypes.
- As in Java, parameters are passed by value. In the example, the value of `y`, 301.4, is copied into `x`. The change to `x` does not affect `y`.
- As in Java, if the function does not return any value, use `void` as the return type.
- Parameters and local variables of functions behave like those in Java.

Recursive Functions

Recursion is essentially the same as in Java.

The only difference is if you have mutually recursive functions, also called **indirect recursion**: for instance, if function A calls function B, while B calls A.

Then you have a problem with the requirement that functions be defined before they are used.

You can get around this problem with function **prototypes**, which just give the signature of the function (return type, function name, and list of arguments, without the code body).

You first list the prototype(s), then you can give the actual function definitions.

Global Variables and Constants

C also provides **global variables**.

- A global variable is defined *outside* all functions, including `main`.
- A global variable can be used inside *any* function in the *same file*.

Generally, global variables that can be changed are frowned upon, as contributing to errors. However, global variables are very appropriate for **constants**. Constants are defined using **macros**:

```
#include <stdio.h>
#define INTEREST_RATE .06
main () {
  double principal = 10000.;
  printf("Amount of interest is %f.\n",
          INTEREST_RATE * principal);
}
```

The `#define` defines a macro, meaning that the text "`INTEREST_RATE`" is to be substituted with the text ".06" *at compile time*.

Boolean Expressions

- The operators to compare two values are the same as in Java: `==`, `!=`, `<`, etc.
- However, instead of returning a boolean value, they return either 0 (means false) or 1 (means true).
- Actually, C interprets *any* non-zero integer as true.

Thus the analog in C of a **boolean expression** in Java is any expression that produces an integer. It is interpreted as producing a truth value, by letting 0 mean false and a non-zero value mean true.

As in Java, boolean expressions can be operated on with && (conditional and), `||` (conditional or), and `!` (negation). Some examples:

- `(10 == 3)` evaluates to 0 (false), since 10 does not equal 3
- `!(10 == 3)` evaluates to 1 (true), since $10 \neq 3$
- `!( (x < 4) || (y == 5) )` : if `x` is 10 and `y` is 5, then this evaluates to 0 (false), since `y` is 5

If Statements and Loops

Given the preceding interpretation of "boolean expression", the following statements are the same in C as in Java:

- `if`
- `if-else`
- `while`
- `for`

Since Boolean expressions are essentially integers, you can have a `for` statement like this in C:

```
for (int count = 99; count; count--) {
  ...
}
```

- `count` is initialized to 99;
- the loop is executed *as long as the expression count is non-zero* (remember that non-zero means true, in the context of a boolean expression);
- `count` is decremented by 1 at each iteration.
- This loop is executed 99 times.

Switch

C has a switch statement that is like that in Java:

```
switch ( <integer-expression> ) {
  case <integer-constant-1> :
      <statements-for-case-1>
      break;
  case <integer-constant-2> :
      <statements-for-case-2>
      break;
  ...
  default : <default-statements>
}
```

Don’t forget the break statements!

The integer expression must produce a value belonging to any of the integral data types (various size integers and characters).

Enumerations

This is something neat that Java does not have.

An **enumeration** is a way to give mnemonic names to a group of related integer constants.

For instance, suppose you need to have some codes in your program to indicate whether a library book is checked in, checked out, or lost. Intead of

```
#define CHECKED_IN  0
#define CHECKED_OUT 1
#define LOST 2
```

you can use an **enumeration declaration**:

```
enum {CHECKED_IN, CHECKED_OUT, LOST};
```

The names in the list are matched up with 0, 1, 2, ...

If you want to give specific values, you can do that too:

```
enum { Jan = 1, Feb, Mar, Apr };
```

Using an Enumeration in a Switch Statement

```
int status;
/* some code to give status a value */
switch (status) {
  case CHECKED_IN :
    /* handle a checked in book */
    break;
  case CHECKED_OUT :
    /* handle a checked out book */
    break;
  case LOST :
    /* handle a lost book */
    break;
}
```

Enumeration Data Type

You can give a name to an enumeration and thus create an **enumeration data type**. The syntax is:

```
enum <name-of-enum-type> <actual enumeration>
```

For example:

```
enum book_status { CHECKED_IN, CHECKED_OUT, LOST };
```

Why bother to do this? Because you can then *create variables of the enumeration type*:

```
enum book_status status;
```

`enum book_status` is the type, analogous to `int`, and `status` is the variable name.

And why bother to do this? *To get compiler support to help make sure these variables only take on prescribed values.* For instance, the following will be allowed:

```
status = LOST;
```

but the following will not:

```
status = 5;
```

In fact, some compilers will not even allow

```
status = 0;
```

Type Synonyms

The enumeration type is our first example of a **user defined type** in C.

It's rather unpleasant to have to carry around the word `enum` all the time for this type.

Instead, you can give a name to this type you have created, and subsequently just use that type – without having to keep repeating `enum`. For example:

```
enum book_status { CHECKED_IN, CHECKED_OUT, LOST };
typedef enum book_status BookStatus;
...
BookStatus status;
```

The `typedef` statement causes `BookStatus` to be a synonym for `enum book_status`.

Structures

C also gives you a way to create more general types of your own, as **structures** These are essentially like objects in Java, if you just consider the instance variables. *A structure groups together related data items that can be of different types.*

The syntax to define a structure is:

```
struct <structure-name> {
    <first-variable-declaration>;
    <second-variable-declaration>;
    ...
}; /* don’t forget this semi-colon! */
```

For instance:

```
struct student {
    int age;
    double grade_point;
};
```

Then you can declare a variable whose type is the `student` structure:

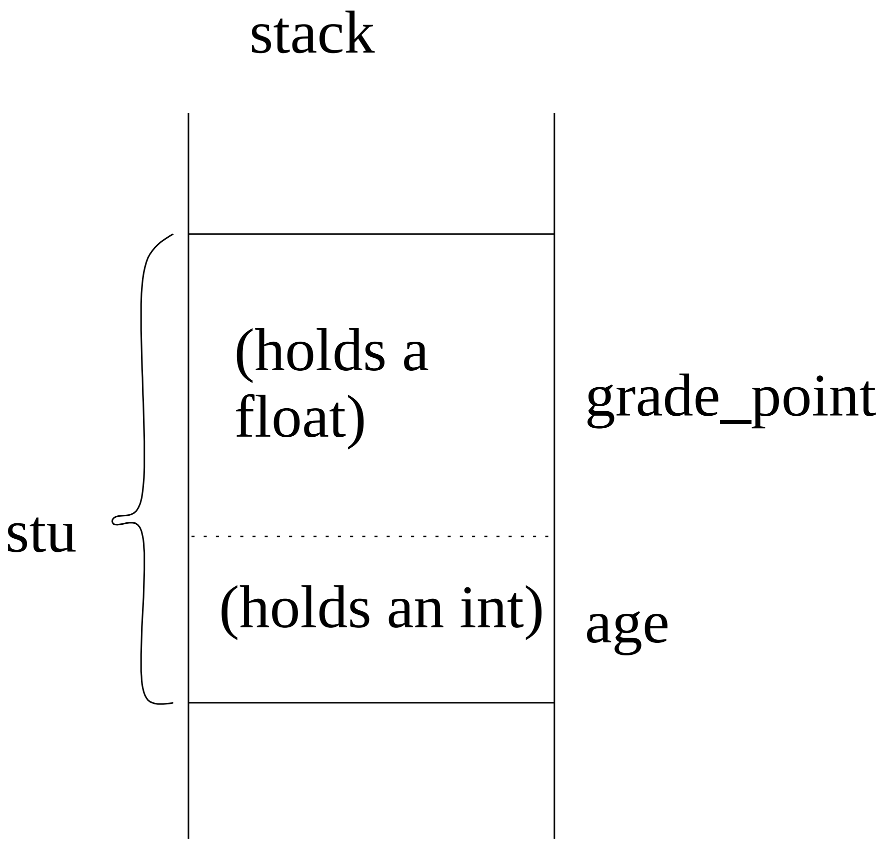
```
struct student stu;
```

Storage on the Stack

The statement

```
struct student stu;
```

causes the entire `stu` structure to be stored *on the stack*:



Using typedef with Structures

When using the structure type, you have to carry along the word `struct`.

To avoid this, you can use a `typedef` to declare a user-defined type, i.e., to provide a name that is a synonym for `struct student`. For instance,

```
struct student {
    int age;
    double grade_point;
};
typedef struct student Student;
```

A more concise way to do this is:

```
typedef struct {
  int age;
  double grade_point;
} Student;
```

Now you can create a `Student` variable:

```
Student stu;
```

Using a Structure

You can access the pieces of a structure using dot notation (analogous to accessing instance variables of an object in Java) :

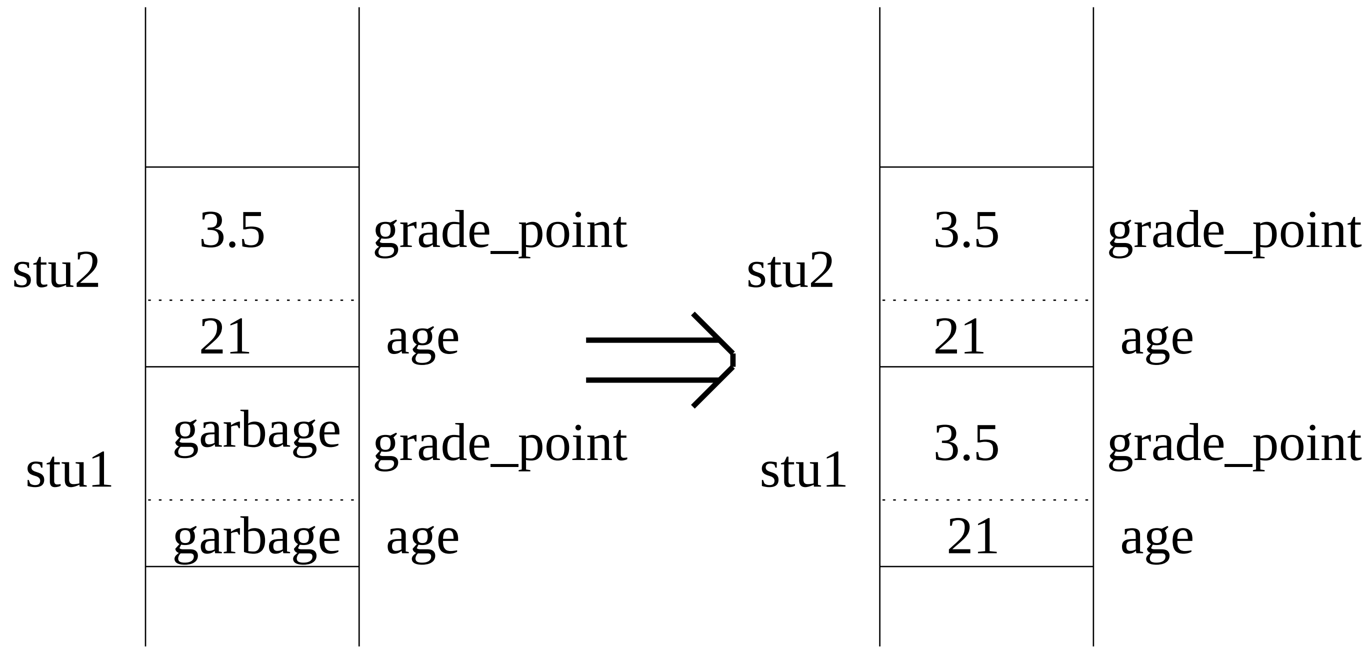
```
stu.age = 20;
stu.grade_point = 3.0;
sum_ages = sum_ages + stu.age;
```

You can also have the *entire* struct on either the left or the right side of the assignment operator:

```
Student stu1, stu2;
stu2.age = 21;
stu2.grade_point = 3.5;
stu1 = stu2;
```

The result is like assigning primitive types in Java! The entire contents of `stu2` are *copied* to `stu1`.

Figure for Copying a Structure



Passing a Structure to a Function

Structures can be passed as parameters to functions:

```
void print_info(Student st) {
    printf("age is %i, GPA is %f.\n",
           st.age, st.grade_point);
    return;
}
```

Then you can call the function:

```
print_info(stu);
```

But if you put the following line of code after the `printf` in `print_info`:

```
st.age = 2 * st.age;
```

the change will NOT be visible back in the main program. You will have only changed the formal parameter copy, not the actual parameter.

Returning a Structure From a Function

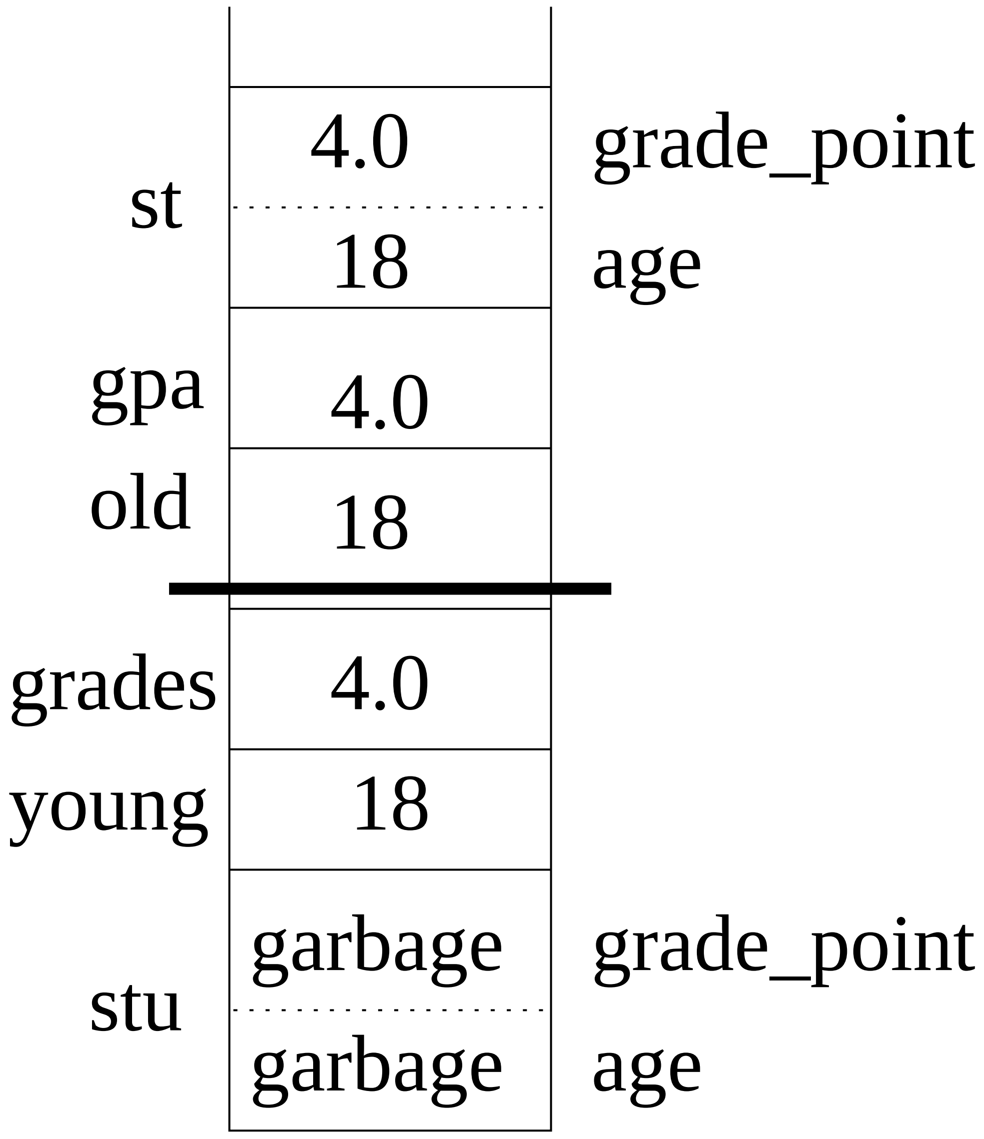
You can return a structure from a function also. Suppose you have the following function:

```
Student initialize
        (int old, double gpa) {
  Student st;  /* local var */
  st.age = old;
  st.grade_point = gpa;
  return st;
}
```

Now you can call the function:

```
Student stu;
int young = 18;
double grades = 4.0;
stu = initialize(young, grades);
```

Figure for Returning a Structure from a Function



The copying of formal parameters and return values can be avoided by the use of pointers, as we shall see.

Arrays

To define an array:

```
<element-type> <array-name> [<size>];
```

For example:

```
int ages[100];
```

- Unlike Java, the size *cannot* go after the element type: `int[100] age` is WRONG!
- Unlike Java, you **must specify the size of the array at compile time**. This means you have to (over) estimate how big of an array you will need. Some of the wasted space can be reduced using pointers, as we will see.
- Unlike Java, all the space for the array is *allocated on the stack!*
- As in Java, numbering of entries begins with 0.
- As in Java, the entries are accessed as `ages[3]`.

Arrays (cont'd)

Two things you CAN do:

- If you have an array of structures, you can access a particular field of a particular entry like this:

  ```
  Student roster[100];
  roster[0].age = 20;
  ```

- You can declare a two-dimensional array (and higher): e.g.,

  ```
  double grades[30][3];
  ```

  As in Java, elements are accessed as `grades[i][j]`.

Two things you CANNOT do:

- **You cannot pass an array as a parameter to a function.**
- **You cannot return an array from a function.**

We'll see how to accomplish these tasks using pointers.

Pointers in C

Pointers are used in C to

- circumvent call-by-value of parameters so that
  - copying of parameters and return values can be avoided
  - lasting changes can be made inside a function
- access array elements in a different way
- allow dynamic memory allocation (e.g., for linked lists)

For each data type T, there is a type which is “pointer to type T”. For instance,

```
int* iptr;
```

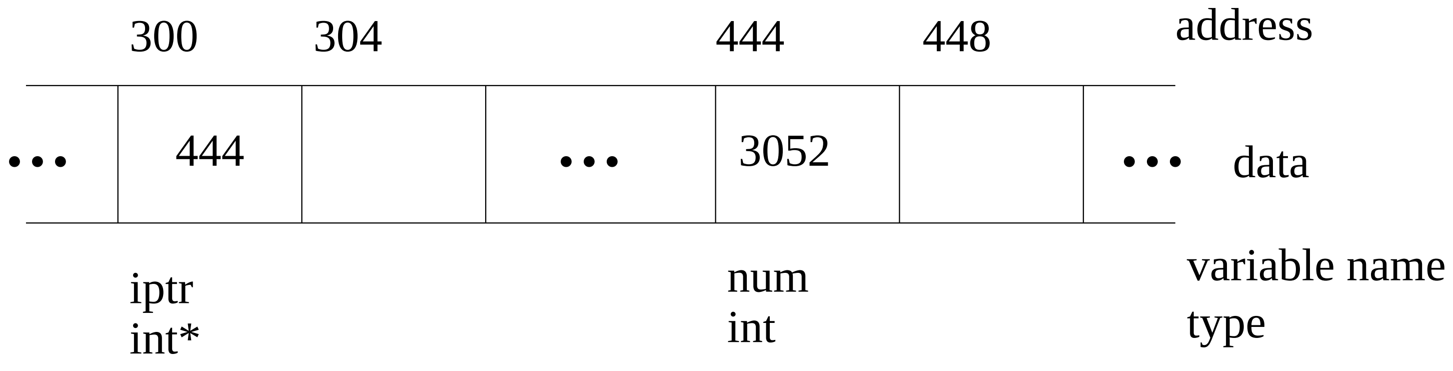
declares `iptr` to be of type “pointer to `int`”. `iptr` refers to a memory location that holds **the address of a memory location that holds an `int`.**
Actually, most C programmers write it as:

```
int *iptr;
```

Addresses and Indirection

Computer memory is a single very long array of bytes.

Each variable is stored in a sequence of bytes.
The **address** of the variable is the index of the starting byte in the array of bytes.



- `iptr` refers to the location of the variable of type `int*` (e.g., 300-303)
- `*iptr` refers to the location whose address is stored in location `iptr` (e.g., 444-447).

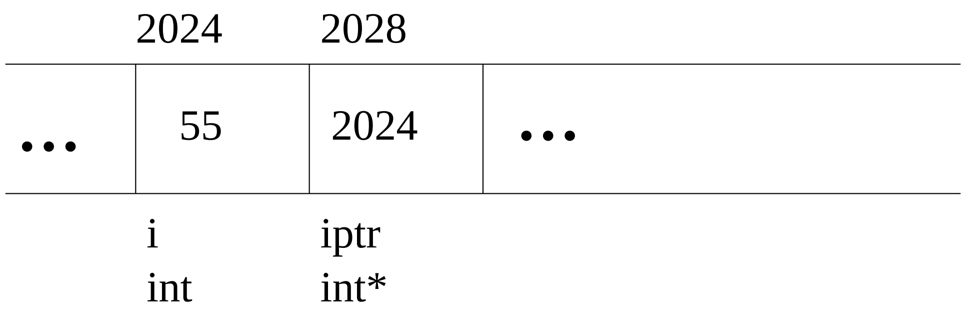
Applying the `*` operator is called **dereferencing** or **indirection**.

The Address-Of Operator

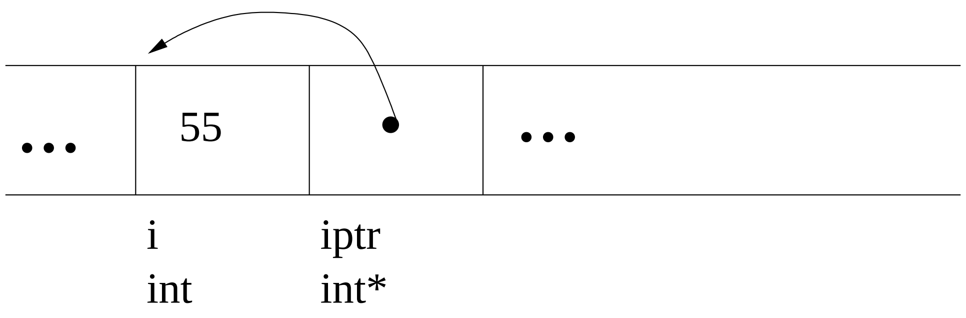
We saw the `&` operator in `scanf`. It *returns the address of the variable to which it is applied.*

```
int i;
int* iptr;
i = 55;
iptr = &i;
*iptr =  *iptr + 1;
```

Last line gets data out of location whose address is in `iptr`, adds 1 to that data, and stores result back in location whose address is in `iptr`.



To abstract away from the numeric addresses:



Comparing Indirection and Address-Of Operators

As a rule of thumb:

- **Indirection:** * is applied to a pointer variable, to refer to the location whose address is stored inside the pointer variable.
  - It CANNOT be applied to non-pointer variables.
  - It CAN appear on either side of an assignment statement.
- **Address-Of:** & is applied to a non-pointer variable, to return the address of the variable.
  - It CAN be applied to a pointer variable.
  - It CANNOT appear on the lefthand side of an assignment statement. (You can't change the address of a variable.)

Pointers and Structures

Remember the struct type `Student`, which has an `int age` and a `double grade_point`:

```
Student stu;
Student* sptr;
sptr = &stu;
```

To access variables of the structure:

```
(*sptr).age
(*sptr).grade_point
```

There is a “shorthand” for this notation:

```
sptr->age
sptr->grade_point
```

Passing Pointer Variables as Parameters

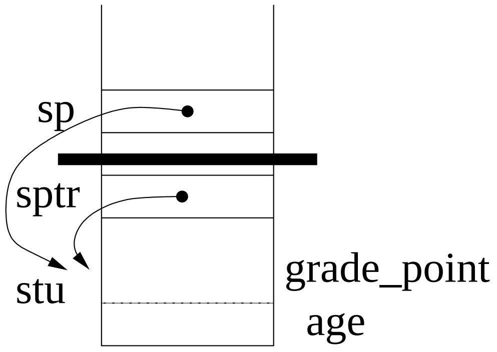
You can pass pointer variables as parameters.

```
void printAge(Student* sp) {
  printf("Age is %i",sp->age);
}
```

When this function is called, the actual parameter must be the *address of a* `Student` *object*:

1. a `Student*` variable: `printAge(sptr);` or
2. apply the & operator to a `Student` variable: `printAge(&stu);`

C still uses call by value to pass pointer parameters, but because they are pointers, what gets copied are *addresses*, not the actual structures. Data coming *in* to the function is not copied.



Passing Pointer Variables as Parameters (cont'd)

Now we can make lasting changes inside a function:

```
void changeAge(Student* sp, int newAge) {
  sp->age = newAge;
}
```

You can also avoid copying data coming *out* of a function by using pointers. Old `initialize` with copying:

```
Student initialize(int old, double gpa) {
  Student st;
  st.age = old;
  st.grade_point = gpa;
  return st;
}
```

More efficient `initialize` using pointers:

```
void initialize(Student* sp, int old, double gpa) {
  sp->age = old;
  sp->grade_point = gpa;
}
```

Passing Pointer Variables as Parameters (cont'd)

Using pointers is an *optimization* in previous case. But it is *necessary* if you want to change two variables, since you can only return one variable:

```
void swapAges (Student* sp1, Student* sp2) {
  int temp;
  temp = sp1->age;
  sp1->age = sp2->age;
  sp2->age = temp;
}
```

To call this function:

```
Student st1, st2;
initialize(&st1, 20, 3.0);
initialize(&st2, 21, 3.1);
swapAges(&st1, &st2);
```

Pointers and Arrays

The name of an array is a pointer to the first element of the array. It is a **constant pointer**, it cannot be changed (for instance with `++`).

```
int a[5];
```

To reference array elements, you can use

- bracket notation `a[0], a[1], a[2], ...`, or
- pointer notation `*a, *(a+1), *(a+2), ...`

What is going on with the pointer notation?

- `a` refers to the address of element 0 of the array
- `*a` refers to the data in element 0 of the array
- `a+1` refers to the address of element 1 of the array
- `*(a+1)` refers to the data in element 1 of the array

This is a SPECIAL meaning of adding 1! It really means add the number of bytes needed to store an `int` (because `a` is a pointer to an `int`).

Pointers and Arrays (cont'd)

You can also refer to array elements with a regular, **non-constant** pointer. For example,

```
int a[5];
int* p;
p = a;     /* p = &a[0]; is same */
```

- `p` refers to the address of element 0 of the array
- `*p` refers to the data in element 0 of the array
- `p+1` refers to the address of element 1 of the array
- `*(p+1)` refers to the data in element 1 of the array

Since `p` is a non-constant pointer, you can also use the increment and decrement operators:

```
for (i = 0; i < 5; i++) {
  *p = 0;
  p++;
}
```

Warning: NO BOUNDS CHECKING IS DONE IN C! Compiler WILL let you refer to `a[5]`.

Passing an Array as a Parameter

To pass an array to a function:

```
void printAllAges(int a[], int n) {
  int i;
  for (i = 0; i < n; i++) {
    printf("%i \n", a[i]);
  }
}
```

The “array” parameter indicates the element type, but NOT the size — *size must be passed separately!* Alternative definition:

```
void printAllAges(int* p, int n) {
  int i;
  for (i = 0; i < n; i++) {
    printf("%i \n", *p);
    p++;
  }
}
```

The *formal* array parameter is a (non-constant) pointer to the element type. You can call the function like this:

```
int ages[4];
printAllAges(ages, 4);
```

Dynamic Memory Allocation in Java

Java does dynamic memory allocation for you. That means that *while your program is running*, memory is assigned to the process as it is needed. This happens whenever you call `new` in Java – the relevant space is allocated.

In Java there is strict distinction between primitive types and object types. Every variable is either of primitive type or object type (i.e., a reference).

- memory for variables is always allocated statically, at the beginning of the block execution, based on the variable declarations. This memory goes away when the surrounding block finishes executing.
- memory for variables of primitive type directly hold the actual contents.
- memory that holds the actual contents of an object is allocated dynamically, whenever `new` is called. This memory goes away after it becomes *inaccessible* (handled by the garbage collector)

Dynamic Memory Allocation in C

In C, there is not the distinction between primitive types and object types. *Every type has the possibility of being allocated statically (on the stack) or dynamically (on the heap).*

To allocate space statically, you declare a variable of the type. Space is allocated when that block begins executing and goes away when it finishes.

To allocate space dynamically, use function `malloc`:

- It takes one integer parameter indicating the length of the space to be allocated. Use `sizeof` operator to get the length; for instance, `sizeof(int)` returns the number of bytes used by an `int` in the current implementation of C.
- It returns a pointer to the beginning of the space. *The space allocated is NOT initialized!* The pointer has type `void*`. You MUST cast it to the appropriate type. If `malloc` fails to allocate the space, it will return `NULL` (a macro for 0.)

`malloc` Example

To dynamically allocate space for an `int`:

```
int* p;
p = (int*) malloc(sizeof(int)); /* cast result
                                   to int* */
if (p == NULL) {      /* to be on the safe side */
  printf("malloc failed!");
} else {
  *p = 33;
  printf("%i", *p);
}
```

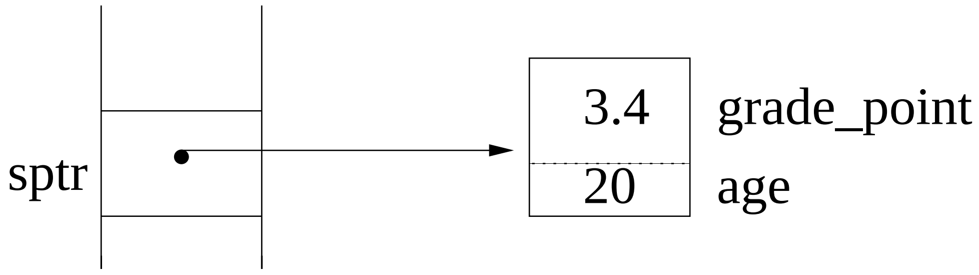
Normally, you don't need to allocate a single integer at a time. Typically, you would use `malloc` to:

- allocate an array at run time, whose size depends on something you've discovered at run time. This could be an array of anything, including structures.
- allocate a structure which will serve as a node in a linked list.

Another `malloc` Example

To dynamically allocate space for a structure:

```
Student* sptr;
sptr = (Student*) malloc(sizeof(Student));
sptr->age = 20;
sptr->grade_point = 3.4;
```



Allocating a Linked List Node Dynamically

For a singly linked list of students, use this type:

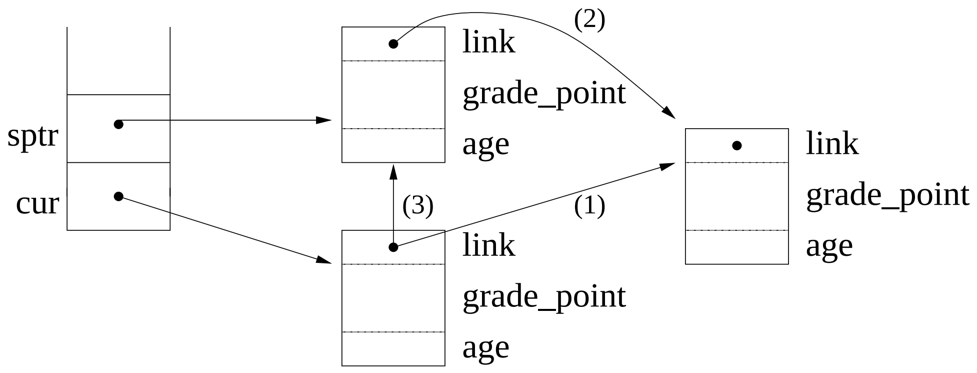
```
typedef struct Stu_Node{
  int age;
  double grade_point;
  struct Stu_Node* link;
} StuNode;
```

To allocate a node for the list:

```
StuNode* sptr;
sptr = (StuNode*) malloc(sizeof(StuNode));
```

To insert the node pointed to by `sptr` after the node pointed to by some other node, say `cur`:

```
sptr=>link = cur->link;
cur->link = sptr;
```



Allocating an Array Dynamically

To allocate an array dynamically, multiply the `sizeof` parameter to `malloc` by the desired number of entries in the array.

```
int i;
int* p;
p = (int*) malloc(100*sizeof(int)); /* 100 elt array */
/* now p points to the beginning of the array */
for (i = 0; i < 100; i++)    /* initialize the array */
  p[i] = 0;                   /* access the elements */
```

Similarly, you can allocate an array of structures:

```
int i;
Student* sptr;
sptr = (Student*) malloc(30*sizeof(Student));
for (i = 0; i < 30; i++) {
  sptr[i].age = 19;
  sptr[i].grade_point = 4.0;
}
```

Deallocating Memory Dynamically

When memory is allocated using malloc, it remains until the program finishes, unless you get rid of it explicitly.

You can get memory leaks because of this:

```
void sub() {
  int *p;
  p = (int*) malloc(100*sizeof(int));
  return;
}
```

Although the space for the pointer variable `p` goes away when `sub` finishes executing, the 100 `int`'s do NOT go away! But they are completely useless after `sub` is done, since there is no way to get to them.

If you had wanted them to be accessible outside of `sub`, you would have to pass back a pointer to them:

```
int* sub() {
  int *p;
  p = (int*) malloc(100*sizeof(int));
  return p;
}
```

Using `free`

To deallocate memory when you are through with it, you call the function `free`. It takes as an argument a pointer to the **beginning** of a chunk of storage that was allocated dynamically, and returns nothing. The result of `free` is that all the space starting at the designated location will be returned to the operating system as available. *The system keeps track of the size of this chunk of storage.*

In the function `void sub` above, just before the return, you should say:

```
free(p);
```

DO NOT DO THE FOLLOWING:

```
int *p;
p = (int*) malloc(100*sizeof(int));
p++;
free(p);  /* BAD! */
```

Now `p` is no longer pointing to the beginning of the allocated space.

Saving Space with Arrays of Pointers

Suppose you need an array of structures, where each structure is fairly large. But you are not sure at compile time how big the array needs to be.

1. Allocate an array of structures that is large enough to handle the worst case.

   Simple but wastes space.

2. Find out at run time how big the array needs to be and then dynamically allocate the space for the array of structures.

   Good if array size does not change.

3. Allocate an array of POINTERS to the structure, that is large enough to handle the worst case.

   Most flexible. The big array consists only of pointers, *which are small. Allocate/deallocate the structs as needed.*

Array of Pointers Example

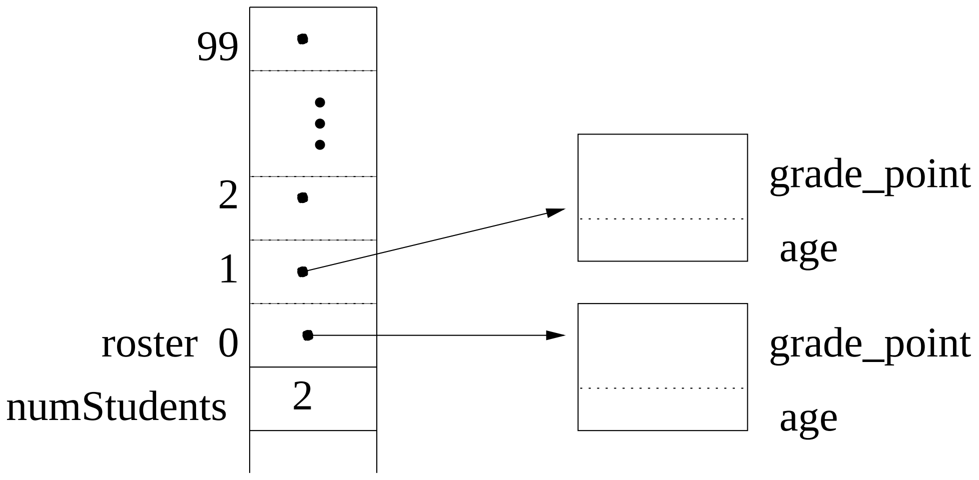
To implement with the usual `Student struct`:

```
int numStudents;
Student* roster[100];

/* code to determine numStudents goes here */

for (i = 0; i < numStudents; i++) {
  roster[i] = (Student*) malloc(sizeof(Student));
  roster[i]->age = 20;
  roster[i]->grade_point = 3.5;
}
```

`numStudents` can be obtained at run time from the user.



Information Hiding in C

Java provides support for information hiding by

- class construct
- visibility modifiers

Advantages of data abstraction, including the use of constructor and accessor (set and get) functions:

- push details out of sight and out of mind
- easier to find where data is manipulated, aiding debugging
- easy to augment what you do with a structure
- easy to improve how data is stored

C does not provide the same level of compiler support as Java, but you can achieve the same effect with some self-discipline.

Information Hiding in C (cont'd)

A "constructor" in C would be a function that

- calls malloc to get the desired space
- initializes the space appropriately
- returns a pointer to the space

For example:

```
Student* constructStudent(int age, double gpa) {
  Student* sptr;
  sptr = (Student*) malloc(sizeof(Student));
  sptr->age = age;
  sptr->grade_point = gpa;
  return stpr;
}
```

Use constructor function to improve readability, maintainability (suppose you want to change how things are initialized), etc:

```
Student* roster[100];
for (i = 0; i < numStudents; i++) {
  roster[i] = contructStudent(20, 3.5);
}
```

Information Hiding in C (cont'd)

The analog of a Java instance method in C would be a function whose first parameter is the "object" to be operated on.

You can write `set` and `get` functions in C:

```
int getAge(Student* sptr) {
  return sptr->age;
}
double getGPA(Student* sptr) {
  return sptr->grade_point
}
void setAge(Student* sptr, int newAge) {
  sptr->age = newAge;
}
void setGPA(Student* sptr, double newGPA) {
  sptr->age = newGPA;
}
```

Information Hiding in C (cont'd)

You can use the `set` and `get` functions to swap the ages for two student objects:

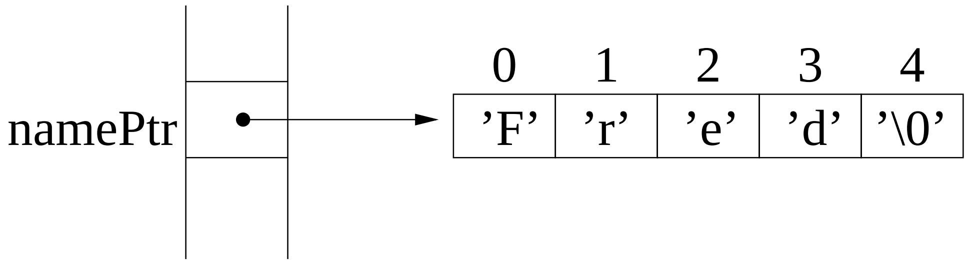
```
Student* s1;
Student* s2;
int temp;
s1 = constructStudent(20, 3.5);
s2 = constructStudent(21, 3.6);
temp = getAge(s2);
setAge(s2,getAge(s1));
setAge(s1,temp);
```

When should you provide set and get functions and when should you not? They obviously impose some overhead in terms of additional function calls. *A good rule of thumb is to provide them when the detailed implementation of a structure might change.*

Strings in C

- There is no explicit string type in C.
- **A string in C is an array of characters that is terminated with the null character.**
- The length of the array is one greater than the number of (real) characters in the string.
- The null character is denoted `'\0'`. It is used to mark the end of the string, instead of storing length information somewhere.
- A sequence of characters enclosed in double quotes causes an array of those characters to be created *on the heap*, ending with the null character.

```
char* namePtr;
namePtr = "Fred";
```



Strings in C (cont'd)

- You can also declare a character array as for any kind of array:

  ```
  char name[20];
  ```

  This declaration causes space to be allocated *on the stack* for 20 chars, and `name` is the address of the start of this space.

  To initialize `name`, *do not assign to a string literal*! Instead, either explicitly assign to each entry in the array or use a string manipulation function.

- Access elements using the brackets notation:

  ```
  char firstLetter;
  name[3] = ’a’;
  firstLetter = name[0];
  namePtr[3] = ’b’;
  firstLetter = namePtr[0];
  ```

Passing Strings to and from Funtions

To pass a string into a function or return one from a function, you must use pointers.
Passing in a string:

```
void printString(char* s) {
  printf("Input is %s", s);
}
```

Notice the use of `%s` in the `printf` statement. The matching data must be a `char` pointer.

Returning a string:

```
char* getString() {
  return "Gig ’em!";
}
```

You can call these functions like this:

```
char* str;
str = getString();
printString(str);
```

Reading in a String from the User

To read in a string from the user, call:

```
scanf("%s", name);
```

- Notice the use of `%s` in `scanf`. The corresponding data must be a `char` pointer.
- `scanf` reads a string from the input stream up to the first whitespace (space, tab or carriage return).
- The letters are read into successive locations, starting with `name`, and then `'\0'` is put at the end.
- *You must make sure that you have a large enough array to hold the string.* How much space is needed? The number of characters in the string plus 1 (for the null character at the end).
- If you don't have enough space, whatever follows the array will be overwritten, so BE CAREFUL!

String Manipulation Functions

There are some useful string manipulation functions provided for you in C. These include:

- `strlen`, which takes a string as an argument and returns the length of the string, *not counting the null character at the end*. I.e., it counts how many characters it encounters before reaching `'\0'`.
- `strcpy`, which takes two strings as arguments and copies its *second* argument to its *first* argument.

First, to use them, you need to include headers for the string handling library:

```
#include <string.h>
```

To demonstrate the use of `strlen` and `strcpy`, suppose you want to add a `name` component to the `Student` structure and change the constructor so that it asks the user interactively for the name:

String Manipulation Functions Example

```
typedef struct {
  char* name;
  int age;
  double grade_point;
} Student;

Student* constructStudent(int age, double gpa) {
  char inputBuffer[100];  /* read name into this */
  Student* sptr;
  sptr = (Student*) malloc(sizeof(Student));
  sptr->age = age;
  sptr->grade_point = gpa;
/* here’s the new part: */
  printf("Enter student’s name: ");
  scanf("%s", inputBuffer);
/* allocate just enough space for the name */
  sptr->name = (char*) malloc (
        (strlen (inputBuffer) + 1)*sizeof(char) );
/* copy name into new space */
  strcpy (sptr->name, inputBuffer);
  return sptr;
}
```

When constructor returns, `inputBuffer` goes away. Space allocated for `Student` object is an `int`, a `double` and just enough space for the actual `name`.

Other Kinds of Character Arrays

Not every character array has to be used to represent a string. You may want a character array that holds all possible letter grades, for instance:

```
char grades[5];
grades[0] = ’A’;
grades[1] = ’B’;
grades[2] = ’C’;
grades[3] = ’D’;
grades[4] = ’F’;
```

In this case, there is no reason for the last array entry to be the null character, and in fact, it is not.

File Input and Output

File I/O is much simpler than in Java.

- Include `stdio.h` to use built-in file functions and types
- Declare a pointer to a type called FILE (provided by the system)
- Call `fopen` to open the file for reading or writing — returns pointer to the file
- Writing to a file is done with `fprintf` (analogous to `printf`.
- Reading from a file is done with `fscanf` (analogous to `scanf`.
- Call `fclose` to close the file.

File I/O Example

```
/* to use the built in file functions */
#include <stdio.h>
main () {
/* create a pointer to a struct called FILE; */
/* it is system dependent */
  FILE* fp;
  char line[80];
  int i;
/* open the file for writing */
  fp = fopen("testfile", "w");
/* write into the file */
  fprintf(fp,"Line %i ends \n", 1);
  fprintf(fp,"Line %i ends \n", 2);
/* close the file */
  fclose(fp);
/* open the file for reading */
  fp = fopen("testfile", "r");
/* read six strings from the file */
  for (i = 1; i < 7; i++) {
    fscanf(fp,"%s", line);
    printf("got from the file: %s \n", line);
  }
/* close the file
  fclose(fp);
}
```

Motivation for Stacks

Some examples of *last-in, first-out* (LIFO) behavior:

- Web browser's "back" button retraces your steps in the reverse order in which you visited the sites.
- Text editors often provide an "undo" mechanism that will cancel editing changes, starting with the most recent one you made.
- The most recent pending method/function call is the current one to execute.
- To evaluate an arithmetic expression, you need to finish evaluating the current sub-expression before you can finish evaluating the previous one.

A **stack** is a sequence of elements, to which elements can be added (**push**) and removed (**pop**): elements are removed in the *reverse* order in which they were added.

Specifying an ADT with an Abstract State

We would like a specification to be as independent of any particular implementation as possible.

But since people naturally think in terms of state, a popular way to specify an ADT is with an abstract, or high-level, “implementation”:

1. describe an abstract version of the state, and
2. describe the effect of each operation on the abstract state.

Specifying the Stack ADT with an Abstract State

1. A stack's state is modeled as a sequence of elements.
2. Initially the state of the stack is the empty sequence.
3. The effect of a push(x) operation is to append x to the end of the sequence that represents the state of the stack. This operation returns nothing.
4. The effect of a pop operation is to delete the last element of the sequence that represents the state of the stack. This operation returns the element that was deleted. If the stack is empty, it should return some kind of error indication.

Specifying an ADT with Operation Sequences

But a purist might complain that a state-based specification is, implicitly, suggesting a particular implementation. To be even more abstract, one can specify an ADT *simply by the allowable sequences of operations*.

For instance:

- push(a) pop(a): allowable
- pop(a): not allowable since stack is empty initially
- push(a) push(b) push(c) pop(c) pop(b) push(d) pop(d): allowable
- push(a) push(b) pop(a): not allowable since a is no longer the top of the stack

But it is more involved to give a precise and complete definition of the allowable sequences without reference to an abstract state.

Additional Stack Operations

Other operations that you sometimes want to provide:

- peek: return the top element of the stack, but do not remove it from the stack; sometimes called top
- size: return number of elements in stack
- empty: tells whether or not the stack is empty

Java provides a `Stack` class, in `java.util`, with methods `push`, `pop`, `empty`, `peek`, and `search`.

Balanced Parentheses

Recursive definition of a sequence of parentheses that is **balanced**:

- the sequence "( )" is <u>balanced</u>.
- if the sequence "s" is <u>balanced</u>, then so are the sequences "s ( )" and "( ) s" and "(s)".

According to this definition:

- ( ) : balanced
- (( ) (( ))) : balanced
- (( ))) ( ) : not balanced
- ( )) (: not balanced

Algorithm to Check for Balanced Parentheses

Key observations:

1. There must be the *same total number* of left parens as right parens.
2. In any prefix, the number of right parens can *never exceed* the number of left parens.

Pseudocode:

```
create an empty stack
for each char in the string
   if char = ( then push ( onto the stack
   if char = ) then pop ( off the stack
   if the pop causes an error then unbalanced
endfor
if stack is empty then balanced else unbalanced
```

Java Method to Check for Balanced Parentheses

Using `java.util.Stack` class (which manipulates objects):

```
import java.util.*;

boolean isBalanced(char[] parens) {
  Stack S = new Stack();
  try {  // pop might throw an exception
    for (int i = 0; i < parens.length; i++) {
      if ( parens[i] == ’(’ )
        S.push(new Character(’(’));
      else
        S.pop(); // discard popped object
    }
    return S.empty();
  }
  catch (EmptyStackException e) {
    return false;
  }
}
```

Checking for Multiple Kinds of Balanced Parens

Suppose there are 3 different kinds of parentheses: (and), [and], { and }.

Modify the program:
When we encounter a), we should pop off a (.
When we encounter a], we should pop off a [.
When we encounter a }, we should pop off a {.

```
boolean isBalanced3(char[] parens) {
  Stack S = new Stack();
  try {
    for (int i = 0; i < parens.length; i++) {
      if (leftParen(parens[i]) // ( or [ or {
        S.push(new Character(parens[i]));
      else {
        char leftp = ((Character)S.pop()).charValue();
        if (!match(leftp,parens[i])) return false;
      }
    }
    return S.empty();
  } // end try
  catch (EmptyStackException e) {
    return false;
  }
}
```

Multiple Kinds of Parentheses (cont’d)

```
boolean leftParen(char c) {
  return ((c == ’(’) || (c == ’[’) || c == ’{’));
}

boolean match(char lp, char rp) {
  if ((lp == ’(’) && (rp == ’)’) return true;
  if ((lp == ’[’) && (rp == ’]’) return true;
  if ((lp == ’{’) && (rp == ’}’) return true;
  return false;
}
```

Postfix Expressions

We normally write arithmetic expressions using **infix** notation: the operator (such as +) goes *in between* the operands (the two numbers being added).

Another way to write arithmetic expressions is to use **postfix** notation: the two operands come first, and the operator comes *after*.
For example,

- `3 4 +` is same as `3 + 4`
- `1 2 - 5 - 6 5 / +` is same as `((1 - 2) - 5) + (6 / 5)`

One advantage of postfix is that *you don't need parentheses to indicate the order of evaluation*.
For instance,

- `(1 + 2) * 3` becomes `1 2 + 3 *`
- `1 + (2 * 3)` becomes `1 2 3 * +`

Using a Stack to Evaluate Postfix Expressions

Pseudocode:

```
while input has more tokens do
   if the token is an operand then
      push it on the stack
   if the token is an operator then
      pop operand x off the stack
      pop operand y off the stack
      apply the operator to x and y
      push the result back on the stack
end while
return the value left on the stack
```

This works when all operators are binary (take two operands).

`StringTokenizer` **Class**

Java's `StringTokenizer` class is very helpful to break up the input string into operators and operands — called "parsing".

- Create a `StringTokenizer` object out of the input string. It converts the string into a sequence of tokens which are separated by specified delimiters.
- Use instance method `hasMoreTokens` to test whether there are more tokens.
- Use instance method `nextToken` to get the next token from the input string.
- Second argument to constructor indicates that, in addition to the whitespace characters (blank, newline, tab, and carriage return), the following are also used as delimiters: +, -, *, and /.
- Third argument to constructor indicates that all delimiters are to be returned as tokens.

Java Method to Evaluate Postfix Expressions

```
public static double evalPostFix(String postfix)
                    throws EmptyStackException {
  Stack S = new Stack();
  StringTokenizer parser = new StringTokenizer
                      (postfix, " \n\t\r+-*/", true);
  while (parser.hasMoreTokens()) {
    String token = parser.nextToken();
    char c = token.charAt(0);
    if (isOperator(c)) {
      double y = ((Double)S.pop()).doubleValue();
      double x = ((Double)S.pop()).doubleValue();
      switch (c) {
        case ’+’:
          S.push(new Double(x+y)); break;
        case ’-’:
          S.push(new Double(x-y)); break;
        case ’*’:
          S.push(new Double(x*y)); break;
        case ’/’:
          S.push(new Double(x/y)); break;
      } // end switch
    } // end if
    else if (!isWhiteSpace(c)) // token is operand
      S.push(Double.valueOf(token));
  } // end while
  return ((Double)S.pop()).doubleValue();
}
```

Evaluating Postfix (cont'd)

```
public static boolean isOperator(char c) {
  return ( (c == ’+’) || (c == ’-’) ||
           (c == ’*’) || (c == ’/’) );
}

public static boolean isWhiteSpace(char c) {
  return ( (c == ’ ’)  || (c == ’\n’) ||
           (c == ’\t’) || (c == ’\r’) );
}
```

Does not handle negative numbers in the input: it interprets -3 as the binary minus operator, followed by 3, instead of the unary minus applied to 3.

Does no error checking to see if operands are well-formed, or if the postfix expression itself is well-formed.

Implementing a Stack with an Array

Since Java supplies a `Stack` class, why bother? Basic understanding; other languages.

Idea: As elements are pushed, they are stored sequentially in an array, keeping track of the last element entered. To pop, return the element at the end of the active part of the array.

Issues for Java implementation:

- elements in the array are to be of type `Object`
- throw exception if try to pop an empty stack
- dynamically increase the size of the array to avoid overflow

To handle the last point, we'll do the following:

- initially, the size of the array is, say, 16.
- if array is full and a push occurs, use `new` to create an array twice the size of current array, and copy everything in old array to new array.

Implementing a Stack with an Array in Java

```
class Stack {
  private Object[] A;
  private int next;

  public Stack () {
    A = new Object[16];
    next = 0;
  }
  public void push(Object obj) {
    if (next == A.length) {
    // array is full, double its size
      Object[] newA = new Object[2*A.length];
      for (int i = 0; i < next; i++)      // copy
        newA[i] = A[i];
      A = newA; // old A can now be garbage collected
    }
    A[next] = obj;
    next++;
  }
  public Object pop() throws EmptyStackException {
    if (next == 0)
      throw new EmptyStackException();
    else {
      next--;
      return A[next];
    }
  }
```

Implementing a Stack with an Array in Java (cont'd)

```
  public boolean empty() {
    return (next == 0);
  }

  public Object peek() throws EmptyStackException {
    if (next == 0)
      throw new EmptyStackException();
    else
      return A[next-1];
  }
} // end Stack class

class EmptyStackException extends Exception {

  public EmptyStackException() {
    super();
  }
}
```

Time Performance of Array Implementation

- push: $O(1)$ UNLESS array is full; then it is $O(n)$ plus time for system to allocate space (more later)
- pop: $O(1)$
- empty: $O(1)$
- peek: $O(1)$

Impementing a Stack with a Linked List in Java

Idea: a push causes a new node to be inserted at the beginning of the list, and a pop causes the first node of the list to be removed and returned.

```
class StackNode {
  Object item;
  StackNode link;
}

class Stack {

  private StackNode top; // first node in list, the top

  public Stack () {
    top = null;
  }

  public void push(Object obj) {
    StackNode node = new StackNode();
    node.item = obj;
    node.link = top;
    top = node;
  }
```

Implementing a Stack with a Linked List in Java (cont'd)

```
  public Object pop() throws EmptyStackException {

    if (top == null)
      throw new EmptyStackException();
    else {
      StackNode temp = top;
      top = top.link;
      return temp.item;
    }
  }

  public boolean empty() {
    return (top == null);
  }

  public Object peek() throws EmptyStackException {
    if (top == null)
      throw new EmptyStackException();
    else
      return top.item;
  }
}
```

Time Performance of Linked List Implementation

- push: $O(1)$ plus time for system to allocate space (more later)
- pop: $O(1)$
- empty: $O(1)$
- peek: $O(1)$

Interchangeability of Implementations

If you have done things right, you can:

- write a program using the built-in `Stack` class
- compile and run that program
- then make available your own `Stack` class, using the array implementation (e.g., put `Stack.class` in the same directory
- WITHOUT CHANGING OR RECOMPILING YOUR PROGRAM, run your program — it will use the local `Stack` implementation and will still be correct!
- then replace the array-based `Stack.class` file with your own linked-list-based `Stack.class` file
- again, WITHOUT CHANGING OR RECOMPILING YOUR PROGRAM, run your program — it will use the local `Stack` implementation and will still be correct!

Motivation for Queues

Some examples of *first-in, first-out* (FIFO) behavior:

- waiting in line to check out at a store
- cars on a street waiting to go through a light
- making a phone call – calls are handled by the phone system in the order they are made

A **queue** is a sequence of elements, to which elements can be added (**enqueue**) and removed (**dequeue**): elements are removed in the *same* order in which they were added.

Specifying the Queue ADT

Using the abstract state style of specification:

- The state of a queue is modeled as a sequence of elements.
- Initially the state of the queue is the empty sequence.
- The effect of an enqueue(x) operation is to append x to the end (**rear** or **tail**) of the sequence that represents the state of the queue. This operation returns nothing.
- The effect of a dequeue operation is to delete the first element (**front** or **head**) of the sequence that represents the state of the queue. This operation returns the element that was deleted. If the queue is empty, it should return some kind of error indication.

Specifying the Queue ADT (cont'd)

Alternative specification using allowable sequences would give some rules (an "algebra"). Some specific examples:

- enqueue(a) dequeue(a): allowable
- dequeue(a): not allowable, since the queue is initially empty
- enqueue(a) enqueue(b) enqueue(c) dequeue(a) enqueue(d) dequeue(b): allowable
- enqueue(a) enqueue(b) dequeue(b): not allowable, a should be returned, not b

Other popular queue operations:

- peek: return the front (head) element of the queue, but do not remove it from the queue
- size: return number of elements in queue
- empty: tells whether or not the queue is empty

Applications of Queues in Operating Systems

The text discusses some applications of queues in operating systems:

- to buffer data coming from a running process going to a printer: the process can typically generate data to be printed much faster than the printer can print it, so the data is saved in order in a “print queue”
- a printer may be shared between several computers that are networked together. Jobs running concurrently may all want to access the printer. Their print jobs need to be queued up, to prevent them from colliding; only one at a time can be printed.

Application of Queues in Discrete Event Simulators

A **simulation program** is a program that mimics, or "simulates", the behavior of some complicated real-world situation, such as

- the telephone system
- vehicular traffic
- weather

These systems are typically too complicated to be modeled exactly mathematically, so instead, they are simulated: events take place in them according to some random number generator. For instance,

- at random times, new calls are placed or some existing calls finish
- at random times, some more cars enter the streets
- at random times, some turbulence occurs

Some of these situations are particularly well described using queues; they are characterized by entities that are stored in queues while waiting for service, for instance, the telephone system and traffic.

Using a Queue to Convert Infix to Postfix

First attempt: Assume infix expression is fully parenthesized.
For example:

- $(((22/7) + 4) * (6 - 2))$
- $(7 - (((2 * 3) + 5) * (8 - (4/2))))$

Pseudocode:

```
create queue Q to hold postfix expression
create stack S to hold operators not yet
       added to postfix expression
while there are more tokens do
   get next token t
   if t is a number then enqueue t on Q
   else if t is an operator then push t on S
   else if t is ( then skip
   else if t is ) then pop S and enqueue result on Q
endwhile
return Q
```

Converting Infix to Postfix (cont'd)

Examples:

- $(((22/7) + 4) * (6 - 2))$

  Q:

  S:

- $(7 - (((2 * 3) + 5) * (8 - (4/2))))$

  Q:

  S:

Converting Infix to Postfix with Precedence

It is too restrictive to require parentheses around everything.

Instead, **precedence conventions** tell which operations to do first, in the absence of parentheses.

For instance, $4*3+2$ equals $12+2 = 14$, not $4*5 = 20$.

We need to modify the above algorithm to handle operator precedence.

- `*` and `/` have higher precedence
- `+` and `-` have lower precedence
- `(` has lowest precedence (a hack)

Converting Infix to Postfix with Precedence (cont'd)

```
create queue Q to hold postfix expression
create stack S to hold operators not yet
       added to the postfix expression
while there are more tokens do
   get next token t
   if t is a number then enqueue t on Q
   else if S is empty then push t on S
   else if t is ( then push t on S
   else if t is ) then
      while top of S is not ( do
         pop S and enqueue result on Q
      endwhile
      pop S // get rid of ( that ended while
   else // t is real operator and S not empty)
      while prec(t) <= prec(top of S) do
         pop S and enqueue result on Q
      endwhile
      push t on S
   endif
endwhile
while S is not empty do
   pop S and enqueue result on Q
endwhile
return Q
```

Converting Infix to Postfix with Precedence (cont'd)

For example:

- $(22/7 + 4) * (6 - 2)$

  Q:

  S:

- $7 - (2 * 3 + 5) * (8 - 4/2)$

  Q:

  S:

Implementing a Queue with an Array

State is represented with:

- array `A` to hold the queue elements
- integer `head` that holds the index of `A` containing the oldest element (which will be returned by the next dequeue); initially 0
- integer `tail` that holds the index of `A` containing the newest element (the most recent element to be enqueued); initially -1

Operation implementations:

- enqueue(x): `tail++; A[tail]:= x`
- dequeue(x): `head++; return A[head-1]`
- empty: `return (tail < head)`
- peek: `return A[head]`
- size: `return tail - head + 1`

Problem: you will march off the end of the array after very many operations, even if the size of the queue is always small compared to the size of `A`.

Implementing a Queue with a Circular Array

Wrap around to reuse the vacated space at the beginning of the array in a circular fashion, using mod operator `%`.

- enqueue(x):

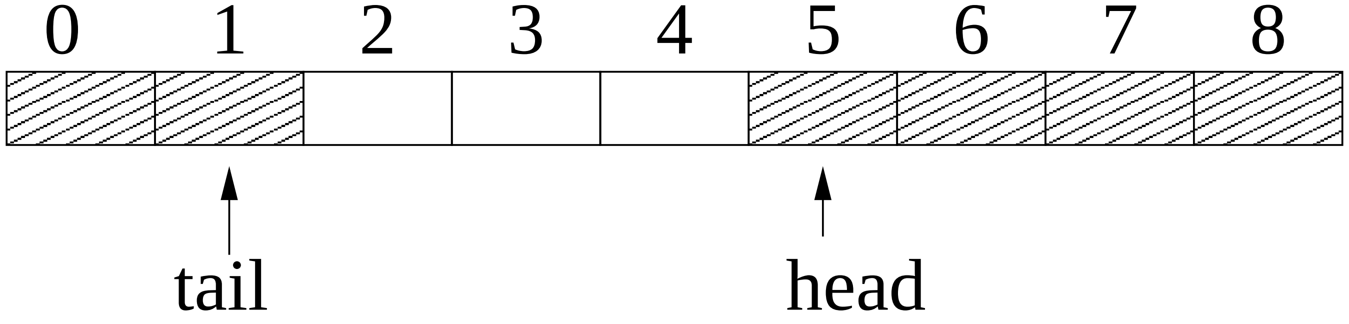
```
tail = (tail + 1) % A.length;
A[tail] = x;
```

- dequeue(x):

```
temp = A[head];
head = (head + 1) % A.length;
return temp;
```

- empty: `return (tail < head);` ???

The problem is that `tail` can wrap around and be in front of `head`, when the queue is not empty.



To get around this problem, add state component:

- integer `count` to keep track of number of elements

Expanding Size of Queue Dynamically

To avoid overflow problem in circular array implementation of a queue, use same idea as for array implementation of stack:
If array is discovered to be full during an enqueue,

- allocate a new array that is twice the size of the old one,
- copy the old array to the new one,
- enqueue new data item onto the new array, and
- free old array (if necessary)

One complication with the queue, though, is that the contents of the queue might be in two sections:

1. from `head` to the end of the array, and
2. then from the beginning of the array to `tail`.

Copying the new array must take this into account.

Performance of Circular Array

Performance of the circular array implementation of a queue:

- Time: All operations take O(1) time, except for enqueue in the case that the array is full
- space: the array allocated might be significantly larger than the size of the queue being represented. Potentially wasteful.

Implementing a Queue with a Linked List

State representation:

- Data items are kept in a linked list.
- Pointer `head` points to the first node in the list, which is the oldest element.
- Pointer `tail` points to the last node in the list, which is the newest element.

Operation implementations:

- To enqueue an item, insert a node containing it at the end of the list, which can be found using the `tail` pointer.
- To dequeue an item, remove the node at the beginning of the list, which can be found using the `head` pointer.

Implementing a Queue with a Linked List (cont'd)

```
class Queue {

  private QueueNode head;
  private QueueNode tail;

  public Queue() {
    head = null;
    tail = null;
  }

  public boolean empty() {
    return (head == null);
  }

  public void enqueue(Object obj) {
    QueueNode node = new QueueNode(obj);
    if empty() {
      head = node;
      tail = node;
    } else {
      tail.link = node;
      tail = node;
    }
  }

// continued on next slide
```

Implementing a Queue with a Linked List (cont’d)

```
// continued from previous slide

  public Object dequeue() {
    if ( empty() )
      return null; // or throw an EmptyQueueException
    else {
      Object returnItem = head.item;
      head = head.link; // remove first node from list
      if (head == null) // fix tail pointer if needed
        tail = null;
      return returnItem;
    }
  }
}
```

Every operation always takes constant time (ignoring time for dynamic memory allocation). No overflow problem and simple code.

Motivation for the List ADT

This ADT is good for modeling a series of data items that have a linear relationship.

Some sample applications:

- the LISP language
- `String`s and `StringBuffer`s in Java
- managing heap storage (aka memory management)

Specifying the List ADT

The *state* of a list object is a sequence of items.

Typical operations on a list are:

- *create:* makes an empty list
- *empty:* returns whether or not the list has no items
- *length:* returns the number of items in the list
- *select(i)*: returns i-th item in list
- *replace(i,x)*: replace i-th item in list with item x
- *delete(x)*: delete item x from the list (or a variant would be *delete(i)*: delete the i-th item)
- *insert(x)*: insert item x into list; variants would be to insert
  - at the front
  - at the end
  - after a particular item
  - before a particular item

Implementing the List ADT

Array implementation:

- Keep a counter indicating the next free index of the array.
- To select or replace at some location, use the random access feature of arrays.
- To insert at some location, shift the later items down.
- To delete at some location, shift the later items up.

Linked list implementation:

- Keep a count of the number of nodes and a pointer to the first node in the list.
- To select, replace, delete or insert an item, traverse the list to get to the appropriate spot.

Comparing the Times of List Implementations

Time for various operations, on a list of n data items:

list operation	singly linked list	array
empty	$O(1)$	$O(1)$
length	$O(1)$	$O(1)$
select(i)	$O(i)$	$O(1)$
replace(i)	$O(i)$	$O(1)$
delete(i)	$O(i)$	$O(n-i)$
insert(i)	$O(i)$	$O(n-i)$

The time for insert in an array assumes no overflow occurs. If overflow occurs, then $O(n)$ time is needed to copy the old array to the new, larger, one.

Comparing the Space of List Implementations

Space requirements:

- If the array holds *pointers* to the items, then there is the space overhead of m pointers, where m is the size of the array allocated.
- If the array holds the items themselves, then there is the space overhead of $m - n$ (unused) items, where n is the current number of items in the list.
- In both kinds of arrays, there is also the overhead of the counter (containing the next free index).
- If you use a linked list, then the space overhead is for n "link" pointers, and the header information.

To quantify the space tradeoffs between the array of items and linked list representations:

- Let p be the number of bytes to store a pointer
- Let q be the number of bytes to store an item
- Let m be the number of elements in the allocated array.

Comparing the Space (cont'd)

To hold n items,

- the array representation uses $q \cdot m$ bytes,
- the linked list representation uses $n \cdot (p + q)$ bytes.

The tradeoff point is when $q \cdot m = n \cdot (p + q)$, that is, when $n = q \cdot m/(p + q)$.

- When $n < q \cdot m/(p + q)$, the linked list is better.
- When $n > q \cdot m/(p + q)$, the array is better.
- When the item size, q, is much larger than the pointer size, p, the linked list representation beats the array representation for smaller values of n.
- When the item size, q, is closer to the pointer size, p, the linked list representation beats the array representation for larger values of n.

Generalized Lists

A **generalized list** is a list of items, where each item might be a list itself.

Example: $(a, b, (c, (d, e), f), g, (h, i))$.

There are five elements in the (top level) list:

1. a
2. b
3. the list $(c, (d, e), f)$
4. g
5. the list (h, i)

Items which are not lists are called **atoms** (they cannot be further subdivided).

Sample Java Code for Generalized List

```
class Node {
  Object item;
  Node link;
  Node (Object obj) { item = obj; }
}
class GenList {
  private Node first;
  GenList() { first = null; }
  void insert(Object newItem) {
    Node node = new Node(newItem);
    node.link = first;
    first = node;
  }
  void print() {
    System.out.print("( ");
    Node node = first;
    while (node != null) {
      if (node.item instanceof GenList)
        ((GenList)node.item).print();
      else S.o.p(node.item);
      node = node.link;
      if (node != null) S.o.p(", ");
    }
    S.o.p(" )");
  }
}
```

Sample Java Code (cont'd)

Notice:

- `o instanceof C` returns true if
  - object o is an instance of class C, or
  - object o implements interface C, or
  - object o is an instance of a subclass of C, or
  - object o is an instance of a subclass of some class that implements interface C
- casts `node.item` to type `GenList`, if appropriate
- recursive call of the `GenList` method `print`
- implicit use of the `toString` method of every class, in the call to `System.out.print`

Don't confuse the `print` method of `System.out` with the `print` method we are defining for class `GenList`.)

Sample Java Code (cont'd)

How do we know that `print` is well-defined and won't get into an infinite loop?

The `print` method is recursive *and* uses a while loop.

- The while loop steps through all the (top-level) items in the current list.
- If an item is not a generalized list, then it simply prints it.
- If an item is itself a generalized list, then the print method recursively calls itself on the current item.
- The while loop stops when you reach the end of the current list.

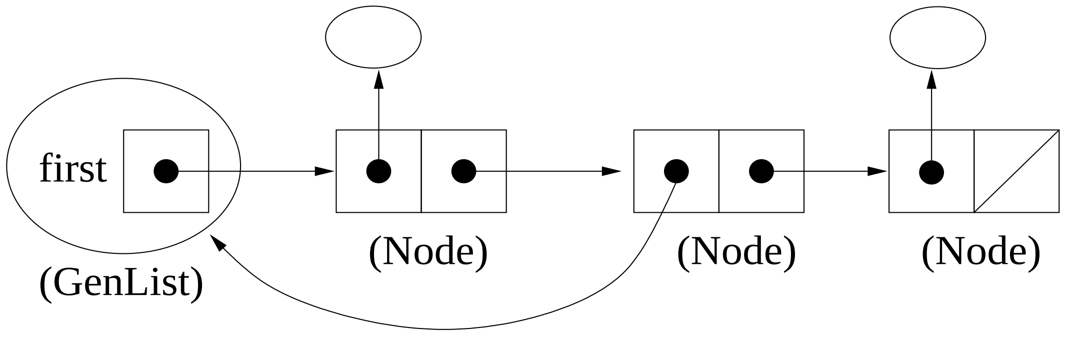
Each recursive call takes you deeper into the nesting of the generalized list.

- Assume there are no repeated objects in the generalized list.
- The stopping case for the recursion is when you reach the most deeply nested list.

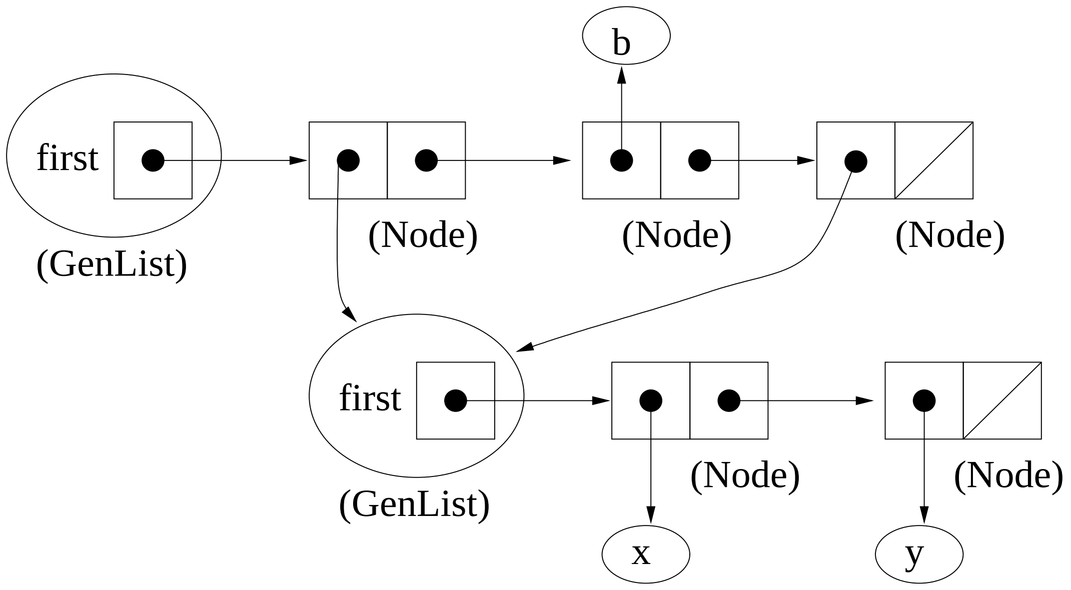
- Each recursive call takes you closer to a stopping case.

Generalized List Pitfalls

Warning! If there is a *cycle* in the generalized list, `print` will go into an infinite loop. For instance:



Be careful about *shared sublists*. For instance,



If you change the first sublist, you will automatically change the second sublist in this case.

Application of Generalized Lists: LISP

Generalized lists are

- highly flexible
- good for applications where data structures grow and shrink in highly unpredictable ways during execution.
- the key structuring paradigm in the programming language LISP (LISt Processing language).

LISP is a **functional** language: every statement is a function, taking some arguments and producing a result.

Each function call is represented as a list, with the name of the function coming first, and the arguments coming after it:

```
( FUNCTION ARG1 ARG2 ... )
```

Each argument could itself be the result of invoking some other function with its own list of arguments, etc.

LISP-like Approach to Arithmetic Expressions

Apply this approach to evaluating arithmetic expressions:

Use **prefix notation** (as opposed to postfix), with parentheses to delimit the sublists:

```
( * (+ 3 4) (+ 8 6) )
```

is equal to `(3 + 4) * (8 + 6)`.

Using the parentheses is useful if we want to allow different numbers of arguments. For instance, let plus have more than 2 arguments:

```
( * (+ 3 4 5) (+ 8 6) )
```

Strings and StringBuffers

Java differentiates between `String`s, which are immutable (cannot be changed) and `StringBuffer`s, which are mutable (can be changed). They are both a kind of list.

There are *no methods that change* an existing `String`.

If you want to change the characters in a string, use a `StringBuffer`. Some key features are:

- change a character at a particular index in the string buffer
- append a string at the end of a string buffer
- insert a string somewhere in the middle of a string buffer

The `StringBuffer` class can be implemented using an array of characters. The ideas are not complicated. You just have to create new arrays and do copying at appropriate times, so it is not particularly fast to do these operations. See Section 7.5 of Standish.

The Heap

When you use `new` or `malloc` to dynamically allocate some space, the run-time system handles the mechanics of actually finding the required free space of the necessary size.

When you make an object inaccessible (in Java) or use `free` (in C), again the run-time system handles the mechanics of reclaiming the space.

We are now going to look at HOW one could implement dynamic allocation of objects from the heap. The reasons are:

- Basic understanding.
- Techniques are useful in other applications.
- Not all languages provide dynamic allocation, including Fortran 66 and assembler. You can use these ideas to “simulate” it.

What is the Heap?

The **heap** is an area of memory used to store objects that will by dynamically allocated and deallocated.

Memory can be viewed as one long array of memory locations, where the address of a memory location is the index of the location in the array.

Thus we can view the heap as a long array of bytes.

Contiguous locations in the heap (array) are grouped together into **blocks**. Blocks can be *different sizes*.

When a request arrives to allocate n bytes, the system

- finds an available block of size at least n,
- allocates the n bytes requested from that block, and
- returns the address of the starting byte allocated.

Blocks are classified as either **free** or **allocated**.

Initially, the heap consists of a single, free, block containing the entire array.

Heap Data Structures

Once blocks are allocated, the heap might get chopped up into alternating allocated and free blocks of varying sizes.

We need a way to locate all the free blocks.

This will be done by keeping the free blocks in a *linked list*, called the **free list**.

The linked list is implemented using *explicit array indices* as the "pointers".

Each block has some **header** information, which includes the size of the block and any required "pointers". (For simplicity, we will ignore the space required for the header.)

Allocation

When a request arrives to allocate n bytes, scan the free list looking for a block that is big enough.
There are two strategies for choosing the block to use:

- **first fit**: stop searching as soon as you find a block that is big enough, OR
- **best fit**: find the smallest block that is big enough. If you find a block that is exactly the required size, you can stop then. If no block is exactly the required size, then you have to search the whole free list to find the smallest one that is big enough.

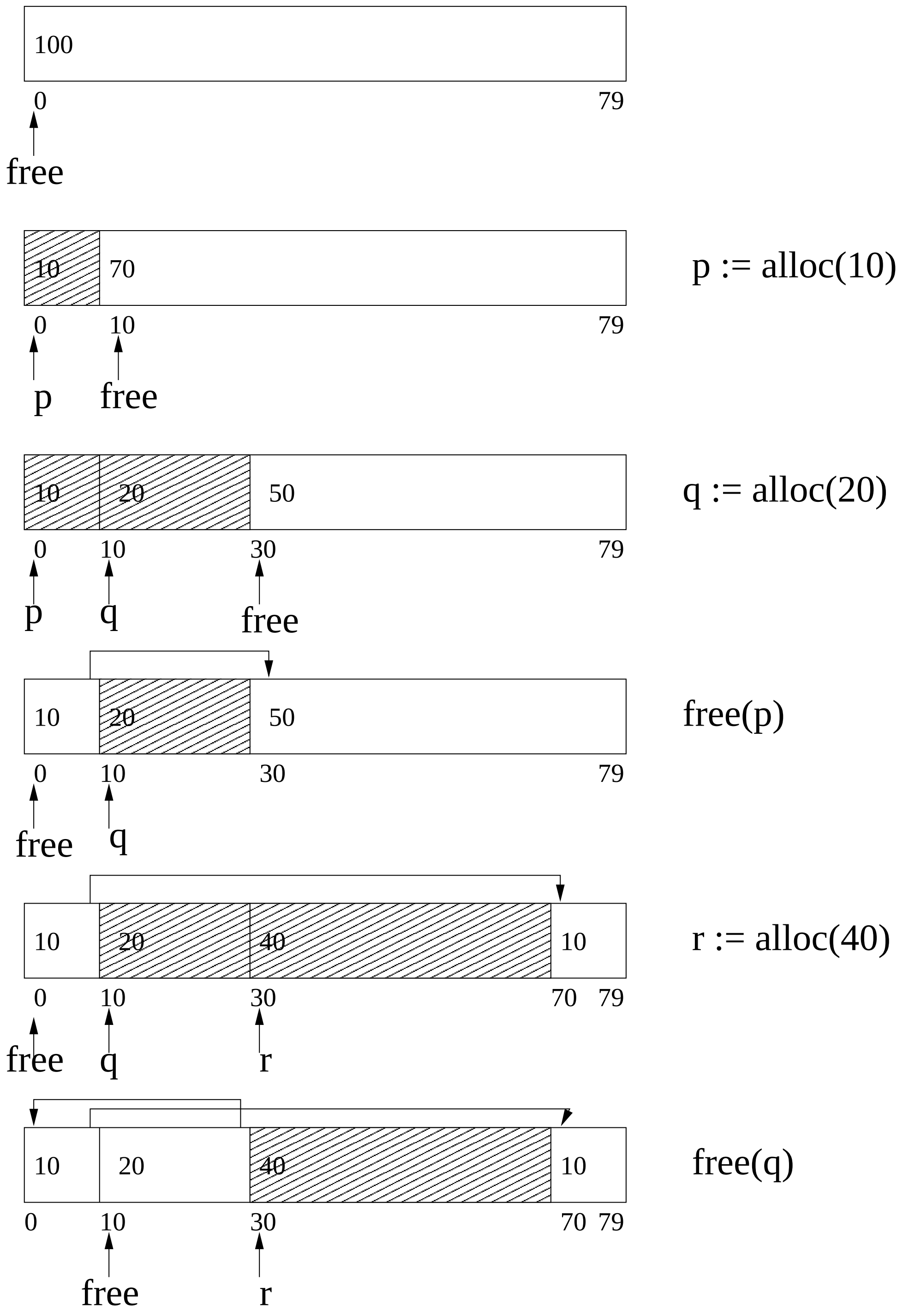
If the block found is bigger than n, then break it up into two blocks, one of size n, which will be allocated, and a new, smaller, free block. The new, smaller, free block will replace the original block in the free list.

If the block found is exactly of size n, then remove it from the free list.

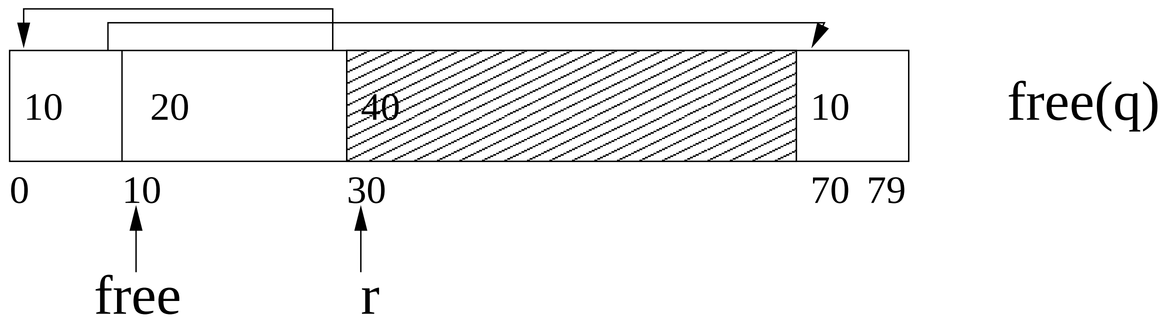
If no block large enough is found, then can't allocate.

Deallocation

When a block is deallocated, as a first cut, simply insert the block at the front of the free list.



Fragmentation



Problem with previous example: If a request comes in for 30 bytes, the system will check the free list, and find a block of size 20, then a block of size 10, and finally a block of size 10. None of the blocks is big enough and the allocation will fail.

But this is silly! Clearly there is enough free space, in fact there are 30 contiguous free bytes! The problem is that the space has been artificially divided into separate blocks due to the past history of how it was allocated.

This phenomenon is called **fragmentation**.

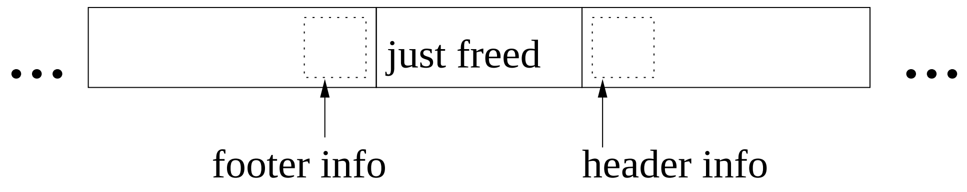
Coalescing

A solution to fragmentation is to **coalesce** deallocated blocks with free (physical) neighbors. Be careful about the use of the word neighbor:

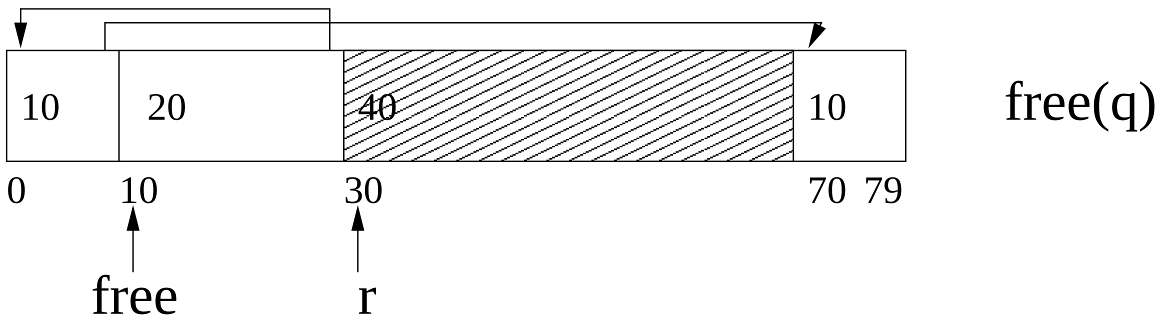
- **physical neighbor:** actual physical space is adjacent
- **virtual neighbor:** blocks are adjacent in the free list, but not necessarily in memory.

To facilitate this operation, we will need additional space overhead in the header, and it will also help to keep "footer" information at the end of each block to:

- make the free list doubly linked, instead of singly linked
- indicate whether the block is free or not
- replicate some info in the footer so that the status of the *physical* neighbors of a newly deallocated node can be efficiently determined



More Insidious Fragmentation



However, coalescing will not accommodate a request for 40 bytes. There are 40 free bytes, but they are not physically contiguous.

The problem is that two of the free blocks are interrupted by the allocated block r.

This is a serious problem with allocation schemes, when the sizes requested can be arbitrary.

Large free blocks keep getting chopped up into smaller and smaller blocks, so it gets harder to satisfy large requests, even if there is enough total space available.

Compaction

The solution to this problem is called **compaction**. The concept is simple: move all the allocated blocks together at the beginning of the heap, and compact all the unallocated blocks together into a single large free block.

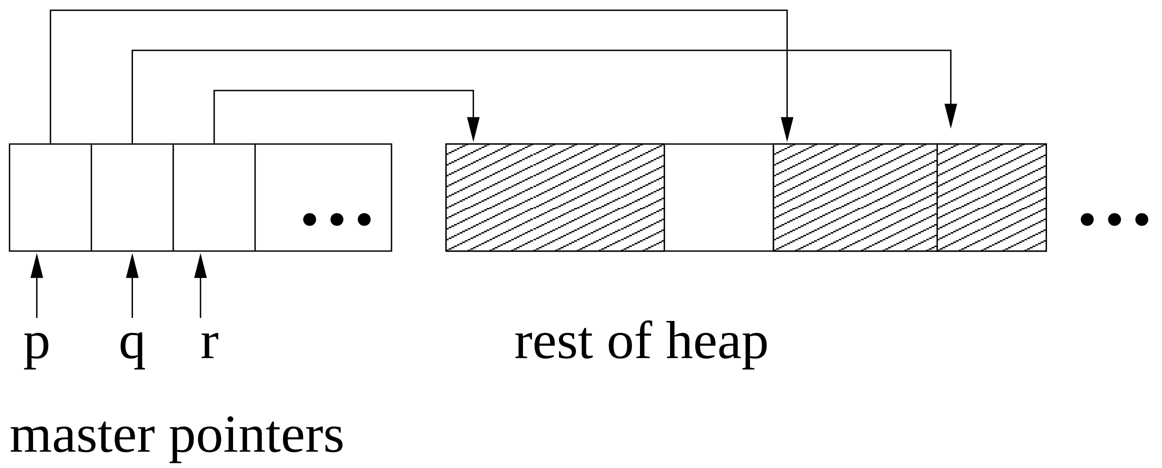
The difficulty though is that if you “move” a block, i.e., copy the information in a block to another location in the heap, you change its address. And you already gave out the original address to the user when the block was allocated!

Master Pointers

A solution is to use **double indirection**, with **master pointers**.

- A special area of the heap contains “master” pointers, which point to (hold the address of) allocated blocks.
- The addresses of the master pointers never change — they are in a fixed part of the heap.
- The address returned by the allocate procedure is the address of the master pointer.
- The *contents* of a master pointer can change, so that when the block being pointed to by a master pointer is moved as part of a compaction, the address is updated in the master pointer.
- But the user, who received the master pointer address, is unaffected.

Master Pointers (cont'd)



Costs:

- Additional space for the master pointers
- Additional time: have to do two pointer dereferences, instead of just one
- Unpredictable "freezing" of execution for a significant period of time, when the compaction occurs. It's hard to predict when compaction will be needed; while it is going on, the application has to pause; and it can take quite a while if the memory is large.

But there really isn't any feasible alternative, if you want to do compaction.

Garbage Collection

The above discussion of deallocation assumes the memory allocation algorithm is somehow informed about which blocks are no longer in use:

- In C, this is done by the programmer, using `free`.
- In Java, the run-time system does this automatically.

This process is part of **garbage collection:**

- identifying inaccessible memory
- management of the free list to reduce the effects of fragmentation

One of the challenging aspects of garbage collection is how to correctly identify inaccessible space, especially how to do it incrementally, so the application does not suddenly pause while it's happening.

There are many interesting algorithms for doing garbage collection with different performance tradeoffs.