EECS 280

Programming and Introductory Data Structures

DeOrio, Juett, Mihalcea, & Olson - Fall 2015

Contributors: Max Smith

Latest revision: August 31, 2015

Contents

Abstract

Techniques and algorithm development and effective programming, top-down analysis, structured programming, testing, and program correctness. Program language syntax and static and runtime semantics. Scope, procedure instantiation, recursion, abstract data types, and parameter passing methods. Structured data types, pointers, linked data structures, stacks, queues, arrays, records, and trees.

1 Introduction

1.1 Compiled vs. Interpreted

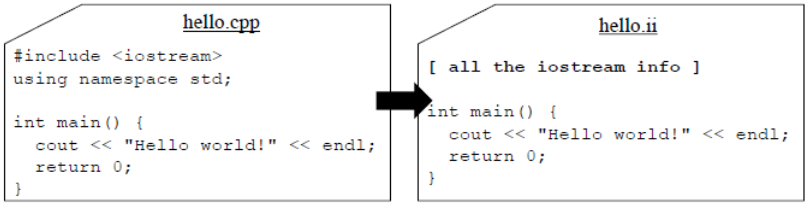
- Language implementation can be compiled or interpreted.
  - **Compiled**: program is converted into low-level machine code before execution.
  - **Interpreted**: program is run step-by-step during execution.

Compiled	Interpreted
Faster	Slower (must go through execution engine)
Less portable (needs recompile)	More portable
Less flexible (complete at compile time)	More flexible

- C++ is a compiled language, for this course we will use **g++** as our compiler
- g++ wraps multiple steps:
  1. Preprocessing
  2. Compilation proper
  3. Assembly
  4. Linking

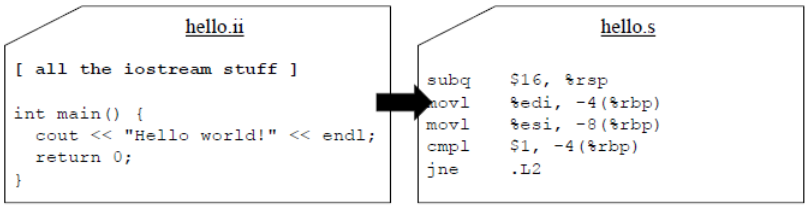
1.2 Preprocessing

- Expands all included libraries (e.g., `#include`, `#define`)
- To view what preprocessing does add the `-E` flag, this will stop it after the preprocessing stage. (e.g., `g++ -E hello.cpp -o hello.ii`)



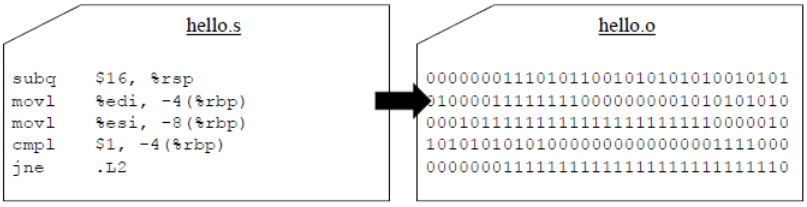
1.3 Compilation proper

- Converts C++ code into assembly instructions
- To view what compilation proper does add the `-S` flag, this will stop it after the compilation proper stage. (e.g., `g++ -S hello.ii -o hello.s`)



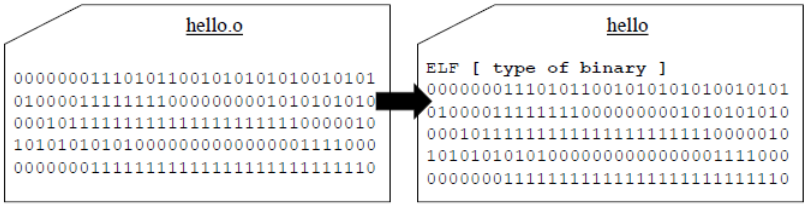
1.4 Assembly

- Converts assembly into binary
- Results in an **object file**
- To view what assembly does add the `-c` flag, this will stop it after the assembly stage. (e.g., `g++ -c hello.s -o hello.o`)



1.5 Linking

- Allows libraries and other object files to find each other
- Creates an executable
- `g++ hello.o -o hello`



1.6 Running the Executable

1. Type program call into shell (`\% ./hello`)
2. Shell asks OS to run program
3. OS loads executable into memory (disk→RAM)
4. OS begins execution
5. Program executes and finishes
6. Control transferred back to OS

2 Procedural Abstraction and Recursion

2.1 Abstraction

- **Abstraction** is a many-to-one mapping that reduces complexity and eliminates unnecessary details by providing only those details that matter.
- e.g., Multiplication: table look-up, summing, etc. (we only care that it multiplies)
- Type types: procedural, data

2.2 Procedural Abstraction

- Decomposing programs into functions
- For any function, there is a person who implements the function (the author) and a person who uses the function (the client).
- The author needs to think carefully about whatthe function is supposed to do, as well as howthe function is going to do it.
- In contrast, the client only needs to consider the what, not the how. Since how is much more complicated, this is a Big Win for the client!
- Provides two important properties:
  - **Local**: the implementation of an abstraction can be understood without examining any other abstraction implementation.
  - **Substitutable**: you can replace one (correct) implementation of an abstraction with another (correct) one, and no callers of that abstraction will need to be modified.
- Layout abstractions before writing code, you don't want to change it!
- If a change can be limited to replacing the implementation of an abstraction with a substitutable implementation, then you are guaranteed that no other part of the project needs to change.

2.3 Fucntion Definitions vs. Declarations

- In larger programs, it is useful to separate the declarationof a function from its definition. A declaration provides the **type signature** of a function, also called the function header.
- Function headers can be placed in their own file and accessed using the preprocessor directive `#include`.
- The **header file** contains all of the function declarations (io.h):

```
1 double GetParam(string prompt, double min, double max);
2 //...
```

- And the associated implementations/definitions (io.cpp):

```
1 #include "io.h"
2 double GetParam(string prompt, double min, double max){
3     /* ... */
4 }
```

- These can be accessed in another file (p1.cpp) through an include of the header file:

```
#include "io.h"
int main() {
    //...
    GetParam(prompt, min, max);
}
```

2.4 Function Details

- All C++ functions take zero or more arguments and return a result of some type.
- There is a special type, called **void**, that means "noresult is returned". void is still a type, even though it is "the type with no legal values".
- The type signature of a function can be considered part of theabstraction if you change it, customers (callers) must also change.
- However, as long as a new implementation of an abstraction does the "same thing" as some old (correct) implementation, you can replace the old one with the new one.
- Now we need to define what "same thing" means through **specifications**

2.5 Specifications

- Answer three questions:
  - **REQUIRES**: What pre-conditions must hold? (if any)
  - **MODIFIES**: Do inputs change? How? (if any)
  - **EFFECTS**: What does it do?
- e.g.,

```
intfactorial(intn);
// REQUIRES: n >= 0
// EFFECTS: returns n!
```

- Functions without REQUIRES clauses are considered **complete**; they are valid for all input. Otherwise they're **partial**.
- Complete is better to write than partial, if possible.
- A MODIFIES clause identifies any function argument or piece of global state that **might** change if this function is called.
- e.g.,

```
void swap(int&x, int&y);
// MODIFIES: x, y
// EFFECTS: exchanges the values of x and y
```

- The ampersand (&) means that you get a **reference** to the caller's argument, not a cop yof it. This lets you to change the value of that argument.

2.6 Call Stacks: How function calls work

- When you call a function, the program follows these steps:
  1. Evaluate the actual arguments to the function (order is not guaranteed).
  2. Create an "activation record" (sometimes called a "stack frame") to hold the function's formal arguments and local variables.
  3. Copy the actuals' values to the formals' storage space.
  4. Evaluate the function in its local scope.
  5. Replace the function call with the result.
  6. Destroy the activation record.
- Activation records are typically stored as a **stack**.
- See example from lecture slides.

2.7 Recursion

- **Recursive**: "refers to itself"
- A function is recursive if it calls itself
- Problem is recursive if:
  1. Base case exists (at least one)
  2. All other cases can be solved by first solving one (or more) smaller cases, and then combining those solutions with a simple step.
- e.g.,

$$n! = \begin{cases} 1 & (n == 0) \\ n \times (n-1)! & (n > 0) \end{cases}$$

```
1 int factorial(int n) {
2     // REQUIRES: n >= 0
3     // EFFECTS: computes n!
4     if (n == 0){
5         return 1; // Base case
6     }else {
7         return n * factorail(n-1); // Recursive step
8     }
9 }
```

- To write a recursive function:
  1. Identify the "trivial" cases, and write the explicitly

2. For all other cases, first assume there is a function that can solve smaller versions of the same problem, then figure out how to get from the smaller solution to the bigger one.

– See example from lecture slides.

3 Tail Recursion

3.1 Another kind of factorial

```
1  int fact_helper(intn, intresult){
2      // REQUIRES: n >= 0
3      // EFFECTS: returns result * n!
4
5      if (n == 0) {
6          return result;
7      }else {
8          return fact_helper(n-1,result*n);
9      }
10 }
11
12 int factorial(intnum){
13     // REQUIRES: num>= 0
14     // EFFECTS: returns num!
15
16     return fact_helper(num, 1);
17 }
```

3.2 Stack effects

– Trace out the stack calls for `factorial(2)` for our new and "old" function:

```
"new"
factorial(2)
-->fact_helper(2, 1)
   -->fact_helper(1, 2)
      -->fact_helper(0, 2)
      <--2
   <--2
<--2
```

```
"old"
factorial(2)
-->2 * factorial(1)
  -->1 * factorial(0)
    <--1
  <--1*1 (==1)
<--2*1 (==2)
```

11

– Effects of the new version on the stack:

  – The activation record of a function is needed only as long as there is computation left over and can be discarded as soon as the return value is known.

  – With the new version, the concrete return value isnt known at the time of the recursive call. However, we do know that whatever that recursive call returns, that will be our return value too.

  – This means that the caller's stack frame isn't needed any more, and we can throw it away.

– This is called **tail recursion**.

- If the result of the recursive call is returned directly with no pending computation, it is tail-recursive. Otherwise, its "plain" recursion.

4 Recursion and Iteration

4.1 Tail-recursion to iteration conversion

- There are five steps to the conversion of a tail-recursive function to an iterative one:

  1. Copy the function's type signature
  2. Identify any needed "loop variables" by inspecting the call to the helper function (if it exists).
  3. Write initialization code to mirror the call to the helper function
  4. Identify termination condition(s) and return values by copying the base case behavior.
  5. Write loop body by copying the inductive step

4.2 Dependency Graphs

- Special kind of graph with "directed" edges showing which "new" values depend on which "old" values.
- An edge is drawn from a **source vertex** to a **sink vertex**.
- To build a dependency graph, draw one vertex for each variable. If variable fooreads from variable barto compute its new value, draw an edge fromfootobar. (Don't draw an edge between a vertex and itself).
- If a variable has no edges with it as a sink(i.e. no edges terminate there), you can write its assignment, and erase it and any edges with it as a source.
- If two variables (`foo, bar`) are each dependent on the other, then you can solve this by inventing shadow variables:

```
1 int foo_new = bar - 1; // Shadow variable
2 int bar_new = foo - 1; // Shadow variable
3 // ------------------
4 foo = bar_new;
5 bar = foo_new;
```

- The transition between these two steps is called a **software epoch** - dependencies do not exist across epochs.

5 Testing and Function Pointers

5.1 Types of testing

- Unit testing
  - Test smaller, less complex, easier to understand units
  - One piece at a time (e.g., a function)
- System testing
  - Test entire project (code base)

  - Do this *after* unit testing
- Regression testing
  - Automatically run all unit and system tests after a code change
- Steps to test:
  1. Understand the specification
  2. Write tests
  3. Check the results

5.2 Understand the Specification

- For an entire assignment, read through the specification very carefully, and make a note of everything it says you have to do
- Required behaviors: what must and must not happen

5.3 Write Tests

- For each of your required behaviors, write one or more test cases that check them.
- To the extent possible, the test case should check exactlyone behavior no more!
- 4 types of tests:
  - **Simple**: "expected" or "normal" inputs
  - **Boundary**: edges of what is expected
  - **Nonsense**: unexpected inputs
  - **Stress**: long inputs

5.4 Check the results

- Do: write down correct answer beforerunning test
- Don't: quickly run test cases and glance at the output
- **Asserts** that the representation invariant is true:

```
1 #include <cassert>
2 list_tin = list_make(); // empty list
3 list_tout = reverse(in); // expect empty list
4 assert( list_isEmpty(out) );
```

5.5 Debugging Strategies

- Read the code
- Remove portions of the code to narrow down error
- Add print statements
- Add `assert()` statements
- Add a more specific test (shorter is better)
- *Use a debugger!*

5.6 Function Pointers

- How do you define a variable that points to a function taking two integers, and returns an integer?

```
1 int (*foo)(int, int);
```

- You read this from "inside out":

```
1 foo                       // foo
2 (*foo)                    // is a pointer
3 (*foo)(    );             // to a function
4 (*foo)(int, int);         // that takes two integers
5 int (*foo)(int, int);     // and returns an integer
```

- Once we've declared foo, we can assign any function to it: `foo = min;`
- And then it can be called as follows: `foo(3, 5)`

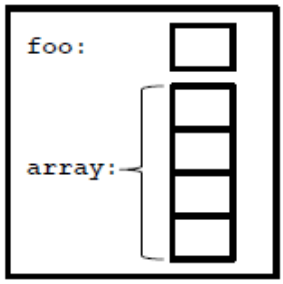
6 Arrays and Pointers

6.1 Arrays

- An **array** is a fixed-sized, indexed, homogeneous aggregate type (a collection of items, all of the same type.)
- To declare and define an array of four integers:

```
1 int bar[4]; // Array
2 int foo;    // For use later
```

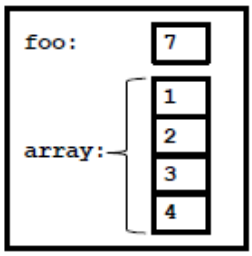
- In the previous code we get a memory environment as follows:



- You can initialize the contents of an array in line:

```
int bar[4] = {1, 2, 3, 4};  // Static Initializer
int foo = 7;
```

- Resulting in the following environment:



6.2 Accessing array elements

- You can access the contents of an array using an "index". The index of the first array element is zero, the next is one, and so on.
- Each individual element is used just like a regular int, so all of the following are legal:

```
// array = {1, 2, 3, 4}
array[1] = 6;
// array = {1, 6, 3, 4}
++array[1];
// array = {1, 7, 3, 4}
array[0] = array[1];
// array = {7, 7, 3, 4}
```

6.3 Passing arrays to functions

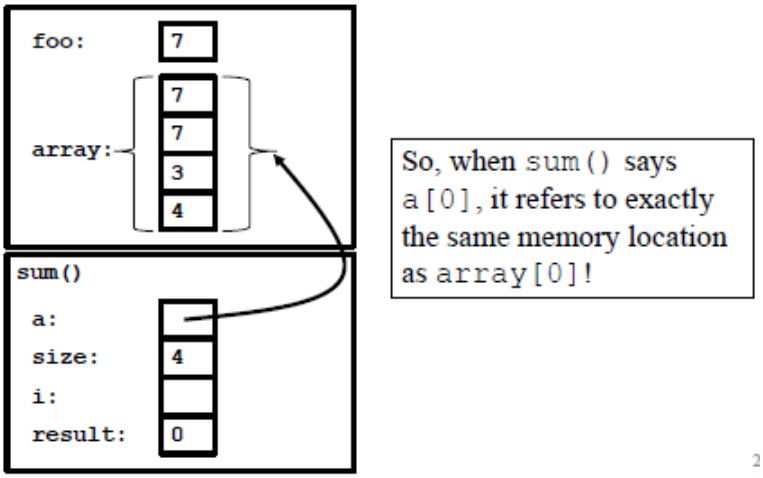
- C++ arrays can be passed as arguments to a function.
- Suppose we wanted to write a function to add up the contents of an array and we want it to work with any size array.
- Here's what the declaration of such a function might look like:

```
int sum(inta[], unsigned int size);
// REQUIRES: there are at least size elements in a[]
// EFFECTS: returns the sum of the first size elements of a[]
```

- Important things to notice:
  1. The declaration of the array argument does not specify the length of the array. That allows the function to work for any length.
  2. `sum()` needs to know how long the array actually is (or at least, how many elements to "sum up"), so the second argument does this.

3. An unsigned intis used for `size` since it cant be negative. A regular intcould be used, but then the REQUIRES clause would need to catch cases of negative sizes. Its better to write a complete function.

– Unlike most types we've seen so far, C++ arrays are notpassed by value. They are passed by reference.



– The C++ compiler DOES NOT CHECK to make sure that your array reference is legal.

– A C-style string is an array of zero or more chars followed by a `NULL` character usually written as `\\0`

```
1     char a[] = "foo"; // {'f', 'o', 'o' '\0'}
```

– **Sentinel**: a special element value for n aggregate type that is:

  – Not a legal value for an element of that aggregate
  – Signals the "end" of the aggregate

6.4 Thinking about memory

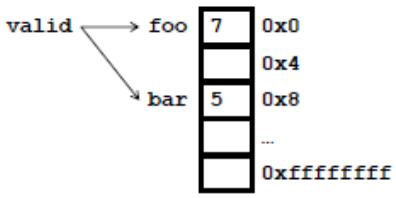
– Memory is a linear sequence of storage locations numbered from 0 to a very large number.

– Locations that hold some sort of object belonging to the program are **valid**, others are not.

– The location of its object is called its **address**.

– e.g.,

```
int foo = 7;
int bar = 5;
```



6.5 Pointers

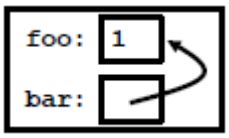
- Sometimes we want to explicitly work with the address of anobject. To do this, we use a **pointer**.
- A "pointer-to-T" is a variable that can hold the address of some other object of type T.
- For example, to declare a pointer to an `int`, we would say:

```
1 int *bar;
```

- The "*" means "pointer-to"
- `bar` is a pointer to `int`

```
1 int foo;
2 int *bar;
3 bar = &foo;
4 foo= 1;
```

- The symbol "&" means "address-of". So, this statement says that "bar is a pointer to an integer, initialized to the address of foo".



6.6 Using pointers

- In addition to setting the value of bar, we can use bar to change the value of the object to which bar points. We do this by saying: `*bar = 2;`
- The "*" here is the "dereference" operator
- For function pointers the compiler allows us to ignore the "address-of" and "dereference" operators

7 Array Traversal

7.1 Passing Pointers to Functions

- Pointers can be arguments to functions. For example, suppose you want a function that adds one to an integer argument passed by reference:

```
1 void add_one(int *x){
2     // MODIFIES: *x
3     // EFFECTS: adds one to *x
4     *x = *x + 1;
5 }
```

- If you were to call this function as so: `add_one(bar);`, where `bar` is a pointer to `foo`

```
int foo;
int *bar;
bar = &foo;

add_one(bar);
```

  - The variable bar is bassed by **value**, but it's a pointer!
  - Both bar and the copy of bar refer to the same address in memory.

- You can also make the call without the "middleman" like: `add_one(&foo);`

7.2 Pointer question

- If you modify `add_one` to:

```
void add_one(int *x){
    x = x + 1;
}
```

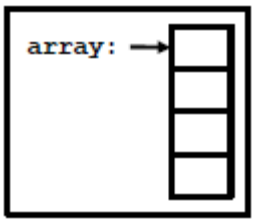
- It will increment the value **of the pointer** by one.
- Pointer arithmetic is done based on units of the **referent type** (the type of the objects in the list).

7.3 Pointers vs. references

- Both allow you to pass objects by reference.
- Pointers require some extra syntax at calling time (&), in the argument list (*), and with each use (*); references only require extra syntax in the argument list (&).
- You can change the object to which a pointer points using arithmetic/assignment, but you cannot change the object to which a reference refers.
- You might wonder why youd ever want to use pointers, since theyrequire extra typing, and allow you to shoot yourself in the foot.
- Why use pointers?
  - Array variables are internally implemented using pointers
  - They allow us to create structures (unlike arrays) whose size is not known in advance; we won't see that use until the last third of the course.

7.4 Pointers and Arrays

- Arrays are actually represented via pointers as so:



- If you were to look at the value of the variable "array" (not `array[0]`) you'd find that it was exactly the same as the address of `array[0]`.
- When the argument `array` is passed to the function sum, a pointer to the first element of the array is really passed and the compiler does all the work of translating something like: `array[3]` into the proper arithmetic/dereference to get the right value.

```
x = array[3];
// Is equivalent to:
int *tmp;
tmp = array + 3;
x = *tmp;
// Or simply:
x = *(array + 3);
```

7.5 Indexing vs. pointer arithmetic

- Using array indexing:

```
for (int i = 0; i < SIZE; ++i){
    cout << array[i] << " ";
}
```

- Using pointer arithmetic:

```
for (int *i = array; i < array+SIZE; ++i){
    cout << *i << " ";
}
```

7.6 Array Traversal Using Pointers

```
int strlen(char *s) {
    char *p = s;
    while (*p) ++p;
    return p - s;
}
```

- `*p` evalues to "false" if `p` points to a NULL, true otherwise.
- `++p` advances by "one character"
- `p-s` computes the "number of characters" between `p` and `s`

7.7 Constants

- `void strcpy(char *dest, const char *src);`
- `const` is a **type qualifier** - something that modifies a type
- It means "you cannot change this value once you have initialized it."
- When you have pointers, there are two things you might change:
  - The value of the pointer.
  - The value of the object to which the pointer points.
- Either (or both) can be made unchangeable:

```
1 const T *p;          // "T" (the pointed-to object) cannot be changed
2 T *const p;          // "p" (the pointer) cannot be changed
3 const T *const p;    // neither can be changed.
```

- Adding `const` will stop changing value mistakes, and the compiler will catch them.
- You can use a pointer-to-T anywhere you expect a pointer-to-const-T, but NOT vice versa
- That's because code that expects a pointer-to-T might try to change the T, but this is illegal for a pointer-to-const-T.
- However, code that expects a pointer-to-const-T will work perfectly well for a pointer-to-T; it's just guaranteed not to try to change it.

7.8 C strings vs. C++ strings

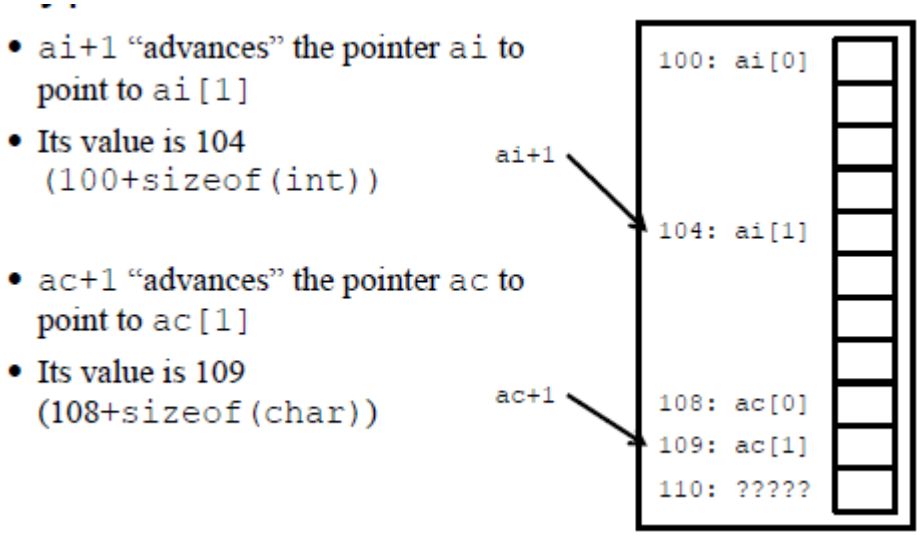
	C string	C++ string
Library headers	`#include <string>`	`#include string`
string constant	`constchar* hello = "hello";`	`conststring hello = "hello";`
length	`strlen(hello);//5`	`hello.length();//5`
local variable	`constintMAXSIZE=1024;` `char s[MAXSIZE];`	`string s;`
copy	`strcpy(s, hello);`	`s = hello;`

concatenate	`constchar* world = " world";`
	`char message[MAXSIZE];`
	`strcpy(message, hello);`
	`strcat(message, world);`
		`string message = hello + " world";`

compare	`if (strcmp(a,b) == 0)` `  // do something`	`if (a == b)` `  // do something`
convert to C++ string	`string cpp_str = hello;`	`char c_str[MAXSIZE];` `strcpy(c_str, message.c_str());`

7.9 Type Sizes

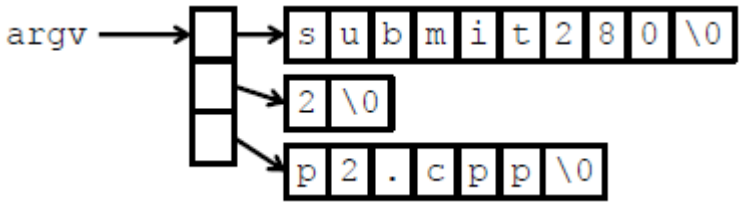
- The amount of memory assigned to a data type is a source of innumerable "portability bugs" in programs.
- There are **some** guarantees, however:
  - A "char" is always one byte
  - A "short" is always at least as big as a char
  - An "int" is always at least as big as a short
  - A "long" is always at least as big as an int
- `sizeof(int)` tells you the number of bytes required to store an `int`



8 Argv, IO, Enums, and Templates

8.1 Argv Basics

- Arguments are passed to programs as an array too `int main(intargc, char *argv[])`
- Since each argument is just a sequence of characters, this array is an array of C-strings: `char *argv[]`



- We also need to know how big the array is: `int argc`
- Suppose we wanted to write a program that is given a list of integers as its arguments, and prints out the sum of that list (you'll need to convert the strings to int):

```
1 intatoi(const char *s);
2 // EFFECTS: parses s as a number and returns its int value
```

8.2 Shell Redirection

- Redirect a file to our program's standard input (cin): `./a.out < inputfile>`
- Redirect our programs' standard output (cout) to a file: `./a.out > outputfile`
- This is not File IO, that is done directly instead of the shell.

8.3 File IO

```
// Open a file using a filestream
string filename = "hello.txt";
ifstreamfilestream;
filestream.open( filename.c_str() );

// CHeck for success opening file
if ( !filestream.is_open() ) {
    cour<< "open failed" << endl;
    exit(1);
}

// Read one word at a time and check that the read was successful
string word;
while ( filestream >> word ) {
    cout<< "word = `" << word << "'\n";
}

// Close file after reading is finished
filestream.close();

/* Alternative: read two words at a time
    string word1, word2;
    while ( filestream>> word1 >> word2 ){
        cout << "word1 = `" << word1 << "'\n"
             << "word2 = `" << word2 << "'\n";
    }
 */

/* Alternative: read numbers
    int i;
    while (filestream >> i) {
        cout << "i = " << i << endl;
    }
 */
```

8.4 Enums

- Enumeration allows for the categorizing of data

```
enum Month {JAN, FEB, MAR, APR, MAY, JUN,
```

```
2               JUL, AUG, SEP, OCT, NOV, DEC};
3 enum Semester {WINTER, SPRING, SUMMER, FALL};
4
5 // Define a variable of the new type:
6 Month this_month;
7 Semester this_semester;
8
9 // Initialize your variables:
10 this_month = MAY;
11 this_semester = SPRING;
```

- Once you have an `enum` type defined, you can use it as a function input or output, just like any other type
- `enum` types are passed by value by default

8.5 Switch statements

- Switch statements work well with enums, and are much faster than long chains of if/else/if/else statements
- Two cases with the same result are grouped together since the arms of the case statement "fall through" to the next `break`

```
1 Semester s = FALL;
2 switch(s) {
3     case SPRING:
4     case SUMMER:
5         cout<< "vacation!\n";
6         break;
7     case FALL:
8     case WINTER:
9         cout<< "work!\n";
10        break;
11    default:  // Add a default, just in case
12        assert(0); // error!
13        exit(1);
14 }
```

8.6 Templates

- The intuition behind templates are that they are code with the "type name" left as a (compile-time) constant.
- The basic idea of templates is that you declare something to be a template, parameterized across one or more types: `template <typename T>`
- Then, write the definition of the function you want templated, using the type `T` where appropriate:

```
1 template <typenameT>
2     T sum(T a[], intsize) {
3     //REQUIRES: T can be initialized to 0 and    +
4     //is defined over T.
```

```
5       T result = 0;
6       for (int i = 0; i<size; ++i)
7           result += a[i];
8       return result;
9   }
```

– Now, we can use this single definition to serve the purposes of both of the "old" functions:

```
1  double a[100] = {... }
2  intb[200] = {... }
3  intc[100] = {... }
4
5  double aSum= sum(a, 100);
6  intbSum= sum(b, 200);
7  intcSum= sum(c, 100);
```

– How it works:

1. When the compiler sees the first call to `sum()`, it inspects the type of the first argument, because that's the "template type".
2. It then generates the code for that version of `sum()`, substituting the word `double` for the type variable `T`. This is called "instantiation".
3. When it sees the second call to `sum()`, it does the same thing by generating a second version of `sum()` and substituting the word `int` for the type variable `T`.
4. When it sees the third call, the compiler knows that that variant of `sum()` already exists and simply reuses it.

9 Structs and Classes

9.1 Structs

– A `struct` describes a "compound object" that comprises one or more elements, each of independent type

– Defines a new type, containing only data

– e.g., a triangle:

```
1  struct Triangle {
2      double a, b, c; // Edge lengths
3  }
4
5  int main() {
6      Triangle t; // New triangle with undefined edges
7
8      // Initialize each member
9      t.a = 3;
10     t.b = 4;
11     t.c = 5;
```

```
12
13      // Or initialize member variables all at once, in order:
14      Triangle t2 = {3, 4, 5};
15
16      // Copy
17      Triangle t3 = t2;
18  }
```

- `struct` are passed by value, unlike arrays
- This can be a problem with large structs, so pass by reference or pointer

9.2 struct vs. array

struct	array
`struct Triangle{/*...*/};`	`double edges[3];`
Stores group of data	Stores group of data
Heterogeneous types	Homogenous types
Access by name	Acces by index
Default pass-by-value	Default pass-by-reference
Creates a custom type	Does not create a new type

9.3 struct vs. class

struct	class
Heterogeneous aggregate data type	Heterogeneous aggregate data type
C style	C++ style
Contains only data	Contains data and functions
Undefined by default	Constructors can be used to initialize
All data is accessible	Control of data access

9.4 Classes

```
1  class Triangle {
2      public:
3      double a,b,c;//edge lengths
4
5      double area(){ //compute area
6          double s = (a+b+c)/2;
7          double newArea= sqrt(s*(s-a)*(s-b)*(s-c));
8          return newArea;
9      }
10 };
```

- A `class` contains both data and functions

- These are called **member data** and **member functions**
- Because member functions are within the same scope as member data, they have access to the member data directly
- `public` means members are accessible from outside the `class`
- From outside scope, access `class` members just like a `struct`

```
int main() {
    Triangle t;
    t.a=3; t.b=4; t.c=5;
    cout << "area = " << t.area() << endl;

    // Copy
    Triangle t2 = t1;
}
```

- Calling a member function is just like evaluating a function

9.5 Initialization problem

- Classes, similar to structs, has undefined values upon initialization.
- This is a common source of bugs.
- A **constructor** is a member function that has the same name as the class
- They run automatically when a new object is defined
- It's typically used to initialize member variables
- A constructor has the same name as the class, and no return value:

```
class Triangle {
public:
    double a, b, c;
    double area() {/*...*/}

    // Constructor
    Triangle() {a=0; b=0; c=0;}

    // Constructors with inputs for initialization
    Triangle(double a_in, double b_in, double c_in){
        a = a_in; b = b_in; c = c_in;
    }
}

int main() {
    Triangle t; // Constructor executes automatically
    t.area();
}
```

- The compiler will select the correct constructor
- Two different functions with the same name, but different prototypes is called **function overloading**.
- There are several ways to call the initializer:

```
int main() {
    Triangle t = Triangle(3, 4, 5);
    Triangle t2(3, 4, 5);
}
```

9.6 Private member variables

- Declaring variables and functions as `private` allows them to be only visible within the scope of that respective class.

```
class Triangle {
private:
    double a, b, c;
public:
    //...
}
```

- In this example, calling `t.a` is no longer possible.
- By default, every member of a class is `private`

9.7 get and set functions

- A `get` function is a `public` function that returns a copy of a `private` member variable:

```
class Triangle {
    //...
public:
    // EFFECTS: returns edge a, b, c
    double get_a() {return a; }
    double get_b() {return b; }
    double get_c() {return c; }
}
```

- A `set` function is a `public` function that modifies a `private` member variable:

```
class Triangle {
    //...
public:
    // REQUIRES: a,b,care non-negative and form a triangle
    // MODIFIES: a, b, c
    // EFFECTS: sets length of edge a, b, c
    void set_a(double a_in);
    void set_b(double b_in);
```

```
 9      void set_c(double c_in);
10  }
```

- `set` functions allow you to run extra code when a member variable changes

9.8 struct vs. class

struct	class
Heterogeneous aggregate data type	Heterogeneous aggregate data type
C style	C++ style
Contains only data	Contains data and functions
Undefined by default	Constructs can be used to initialize
All data is accessible	Control of data access

10 Abstract Data Types (ADTs)

10.1 Abstraction

- **Procedural abstraction**
  - A function is given a specification which documents whatthe function does, but not howit does it.
  - For example, if we find a faster way to compute factorial, we can just replace the old implementation with the new one, and no other component of the program needs to know this.
- **Data abstraction**
  - An ADT provides an abstractdescription of values and operations.
  - The definition of an ADT must combine bothsome notion ofwhatthe values of that type represent, and howthey behave.
  - We can leave off the details of how this actually happens.

10.2 Abstract Data Types

- **Information hiding**: we don't need to know the details of how the object is represented, nor do we need to know how the operations on those objects are implemented.
- **Encapsulation**: the objects and their operations are defined in the same place; the ADT combines both data and operation in one entity.
- ADTs are substitutable: you can change the implementation and no users of the type can tell
- See example in lecture slides.

10.3 const member functions

- You can use `const` in a function signature promises "this member function will not modify any member variable"

```
class Triangle {
    //...
    double area() const;
    //...
}
```

10.4 Representation invariant

- Member variables are a classs representation
- The description of how member variables should behave are representation invariants
- **Representation invariants** are rules that the representation must obey immediately before and immediately after any member function execution
- The *what* the data type does is then the header file
- The *how* the data type works is in the cpp file

10.5 Scope Resolution Operator

- `::` is the scope resolution operator, which tells the compiler that this function is inside the scope of the class
- e.g.,

```
Triangle::Triangle(): a(0), b(0), c(0) {}
```

- In the previous example, it's used to show in-line construction
- This syntax is called an **initializer list**
- More efficient, because dealt with at compile time.
- It can also be used with parameters. e.g.,

```
Triangle::Triangle(double a_in, double b_in, double c_in): a(a_in), b(b_in),
    c(c_in) {}
```

- See the Flatland example from lecture slides.

11 Subtypes and Subclasses

11.1 Subclass

- In Flatland, soldiers and craftsmen have something in common: they are both workers, represented by triangles.
- We can represent this kind of relationship between two C++ classes with a *derived class* (or referred to as: *inherited class* or *subclass*)
- Subclasses contain all of the functionality and data of their parent class

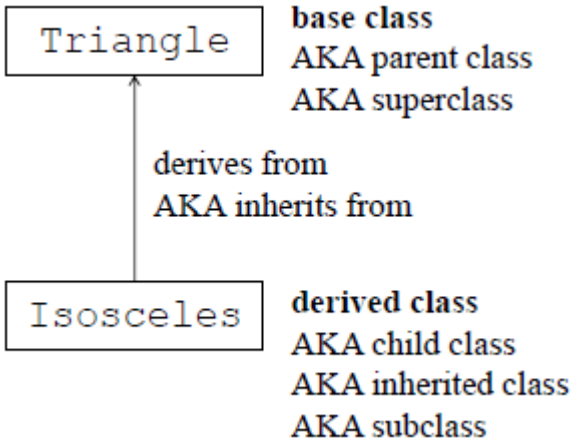
- An "is a" relationship
- e.g.,

```
class Isosceles: public Triangle {
    // ...
}
```

- Isosceles "is a" triangle

11.2 Class hierarchy

- Derivation is often represented by a graph, where each vertex is a class, and each edge shows derivation



11.3 Adding member functions

- Subclasses need to respect the abstraction of their parent classes
- Use getters and setters to deal with private variables

11.4 Derived class constructors

- Constructors **are not** inherited.
- Constructors run automatically, **starting with the base class**.
- e.g.,
  - First, `Triangle` constructor runs
  - Then, `Isosceles` constructor runs
- You can also re-use the parent class constructor:

```
Isosceles(double base, double leg): Triangle (base, leg, leg) {}
```

- But do not call the parent constructor outside the initializer list

- **DO NOT DO THIS:**

```
Isosceles(double base, double leg){
    Triangle(base, leg, leg); // BAD - anonymous object
}
```

11.5 Override vs. Overload

- A function **override** is where a derived class has a function with the same name and prototype as the parent

```
Triangle::set_b(double b_in);
Isosceles::set_b(double b_in);
```

- A function **overload** is where a single class has two different functions with the same name, but different prototypes

```
Triangle::Triangle();
Triangle::Triangle(double a_in,double b_in,double c_in);
```

11.6 protected members

- `protected` members can be seen by all members of this class and any derived classes

```
class Triangle {
public:
    //member functions ...
protected:
    //edge lengths represent a triangle
    double a,b,c;
};
```

- Use the scope resolution operator (::) to call inherited functions: `Triangle::set_b(b_in);`

11.7 Liskov Substitution Principle

- If S is a subtypeof T, then objects of type T may be replaced with objects of type S without altering any of the desirable properties of that program (correctness)
- Or: For any instance where an object of type T is expected, an object of type S can be supplied without changing the correctness of the original computation
- Not all C++ derived classes are subtypes
- For a derived type to also be a subtype, code written to correctly use the supertypemust still be correct if it uses the subtype
-

11.8 How to create a subtype

- With Abstract Data Types, there are three ways to create a subtype from a derived type:
  1. Weaken the precondition of one or more operations
  2. Strengthen the postconditionof one or more operations
  3. Add one or more operations
- 1 & 2 apply to overriden functions:
  - The overridden member function must require no more of the caller than the old method did, but it can require less
  - The overridden member function must do everything the old function did, but it is allowed to do more as well

Weaken precondition

- The preconditions of a method are formed by two things:
  - Its argument type signature
  - The REQUIRES clause
- We can weaken the preconditions by requiring less:

```
1  //REQUIRES: b_inis non-negative and forms a
2  // triangle with existing edges
3  //MODIFIES: this
4  //EFFECTS: sets edges b and c
5  void Isosceles::set_b(double b_in) {
6      Triangle::set_b(b_in);
7      Triangle::set_c(b_in);
8  }
9
10 //Becomes:
11 //REQUIRES: b_inis non-negative andforms a
12 // triangle with existing edges
13 //MODIFIES: this
14 //EFFECTS: sets edges b and c
15 void Isosceles::set_b(double b_in) {
16     b_in= abs(b_in);
17     Triangle::set_b(b_in);
18     Triangle::set_c(b_in);
19 }
```

Strengthen postcondition

- The postconditionsof a method are formed by two things:
  - Its return type signature

  - The EFFECTS clause

- We can strengthen the EFFECTS clause by promising everything we used to, plus extra

```
void Isosceles::set_b(double b_in) {
    Triangle::set_b(b_in); //does everything Triangle did
    Triangle::set_c(b_in); //plus more
}
```

Add an operation

- The final way of creating a subtype is to add a member function
- Any code expecting only the old function will still see all of them, so the new function won't break old code

```
class Isosceles : public Triangle {
public:
    //...
    void set_base(double base){/*...*/}
    void set_leg(double leg) {/*...*/}
};
```

12 Polymorphism

12.1 Static type

- **Static type**: when a compiler can tell which derived type is needed

```
Trianglet(3,4,5);
t.print();

Triangle *t_ptr= &t;
t_ptr->print();

Isoscelesi(1,12);
i.print();

Isosceles*i_ptr= &i;
i_ptr->print();
```

12.2 Dynamic type

- **Dynamic type**: type is not known until run time

```
//EFFECTS: asks user to select Triangle,
// Isosceles or Equilateral
// returns a pointer to correct object
Triangle * ask_user();

```

```
6 int main() {
7     Triangle *t = ask_user(); //enters "Isosceles"
8     t->print();
9 }
```

12.3 Polymorphism

- From the previous example:

```
1 //...
2 Triangle *t = ask_user(); //"Isosceles"
3 t->print();
```

- `t` can change types at runtime, in other words it is **polymorphic**
- Polymorphism is the ability to associate many behaviors with one function name dynamically (at runtime)

12.4 Virtual Functions

- A `virtual` function checks dynamtic type to find correct overridden function version to run.

```
1 class Triangle {
2     virtual void print() const{/*...*/}
3     //...
4 };
```

- The `virtual` function type is inherited, so any sub-classes of `Triangle` that override `print()` will automatically become `virtual`

12.5 Abstract base class

- An **abstract base class** creates an *interface only* class as a base class, from which an implementation can be derived
- You **cannot create an instance** of an abstract base class.
- In C++ a class is abstract if there's missing/no implementation (via having at least one of its functions as **pure virtual** function)

```
1 class Shape {
2 public:
3     virtual double area() const = 0;
4     virtual void print() const = 0;
5 };
```

- "Assigning" a 0 to a function designates as pure virtual
- A **factory function** creates objects for a client who doesn't need to know their actual types (dynamic)

12.6 Upcast

- An **upcast** is substituting a subtype for a supertype
- Type conversion is automatic, an *implicit cast*

```
Shape * ask_user() {
    cout << "Rectangle, Triangle, Isosceles or Equilateral? ";
    string s;
    cin >> s;
    if (s == "Triangle") return &g_triangle;    // Upcasted to shape
    if (s == "Isosceles") return &g_isosceles;  // Upcasted to shape
    //...
}
```

- Since `Isosceles` *is a* `Shape`, this can happen automatically.

12.7 Downcast

- Can't convert supertype to subtype
- e.g., `Shape`→`Isoscelese`
- Type conversion is not automatic
- This is called a *downcast*

```
Isosceles *t= ask_user();//enter "Isosceles"
// Error: invalid conversion
```

- Since a parent type might not be a child type, this cast cannot happen automatically
- `dynamic_cast<T*>(ptr)` downcasts `ptr` to type `T*`

12.8 dynamic_cast vs. static_cast

dynamic_cast	static_cast
Checks at runtime if it is OK to cast	Does not check at runtime
Cast from a polymorphic base to derived	Programmer tells compiler "trust me"
Supertype to subtype, downcast	Non-polymorphic types & polymorphic

13 Container ADTs

- The purpose of a *container* is to hold other objects.
- They need to maintain representation invariants across all actions.
- See lecture slides for example of `IntSet`

14 Interfaces and Invariants

14.1 Interfaces

- Question: How can you provide a class definition that carries no implementation details to the client programmer, yet still has interface information?
- Answer: Create an "interface-only" class as an abstract base class, from which an implementation can be derived.
- Interfaces have **no implementation** which makes it different from an abstract class, which may have partial implementation.
- This is done using pure virtual functions
- e.g., (Notice how all of the functions are pure virtual)

```
1  class IntSet{
2      // OVERVIEW: interface for a mutable set of ints
3      // with bounded size
4
5  public:
6      virtualvoid insert(intv) = 0;
7      virtualvoid remove(intv) = 0;
8      virtualboolquery(intv) const= 0;
9      virtualintsize() const= 0;
10 };
```

- You **cannot** instantiate `IntSet`, because there is no implementation
- You can create a pointer however
- e.g.,

```
1  intmain() {
2      IntSetUnsortedis;
3      IntSet*is_ptr= &is; //pointer to implementation
4      is_ptr->insert(7);
5      is_ptr->insert(4);
6      is_ptr->insert(7);
7      is_ptr->insert(1);
8      is_ptr->remove(4);
9  }
```

14.2 Representation Invariant Details

- A *representation invariant* must be true immediately before and immediately after any member function execution
- With two **exceptions** of course:

  - **Constructors**: the constructor establishes the representation invariant, so the representation invariant only has to hold at the end
  - **Destructors**: the representation invariant only has to hold at the beginning
- You may find it helpful to create a `check_invariant()` function to run tests on whether your functions maintain their representation.

15 Memory Models

15.1 Global variables

- Global variables are defined anywhere outside of a function definition
- Space is set aside for these variables before the program begins execution, and is reserved for them until the program completes
- This space is reserved at compile time
- e.g.,

```
1 const int SIZE=10;  // Global variable
2 int main() {
3     //...
4 }
```

15.2 Local variables

- Local variables are any variable defined within a block
- This includes function arguments, which act as if they were defined in the outermost block of the function
- Space is set aside for these variables when the relevant block is entered, and is reserved for them until the block is exited
- This space is reserved at run time, but the size is known to the compiler
- e.g.,

```
1 int sum(int *array, int size) {
2     int sum=0;  // Local variable, alongside array (the pointer!) and size
3     //...
4 }
```

15.3 Dynamic variables

- It is dynamic because:
  - Size (or number) is determined at runtime
  - When it will be created and destroyed is determined at runtime

new

- Create dynamic variables using new

```
1 int main() {
2     int *p = new int;
3 }
```

- This creates new space for an integer, and returns a pointer to that space, assigning it to `p`
- The initial value is undefined
- Use initializer syntax to assign an initial value:

```
1 int main() {
2     int *p = new int(5);
3 }
```

delete

- Destroy dynamic variables using `delete`

```
1 int *p= new int;
2 //do something with p
3 delete p;
4 delete p; //Error!
```

- Releases the claim on the space previously used by the `int`
- You can only `delete` a dynamic variable *once*

```
1 delete p;
2 delete p; //Error!
```

- Assigning NULL to deleted pointers prevents this issue:

```
1 delete p; p=0;
2 delete p; // Ok
```

- Only objects that are created with `new` can be destroyed by `delete`

Dynamic array

- e.g.,

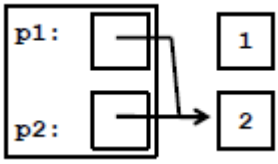
```
//ask user to enter integer size
int size = get_size_from_user();

int *p= new int[size];
//do something with p ...
delete[] p;
```

15.4 Memory leaks

- Dynamic variables live until the programmer destroys them using `delete`
- This is true even if you "forge" the pointer to the object.

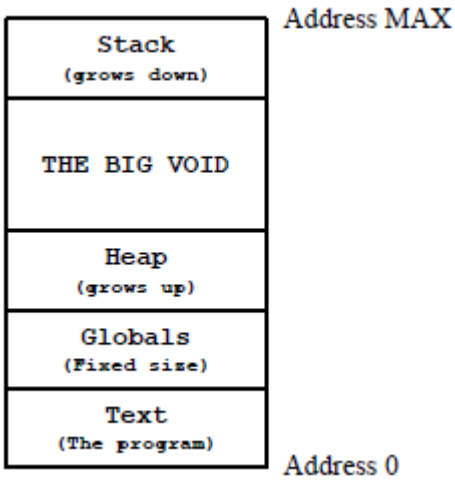
```
1 int main() {
2     int *p1 = new int(1);
3     int *p2 = new int(2);
4     p1 = p2;
5 }
```



- This leaves us with an allocated dynamic object that wehave no means of reclaiming called a **memory leak**

15.5 The heap

- The space for objects created via `new` comes from a location in memory called the heap.
- A running program has an "address space", a collection of memory locations that are accessible to it



- **Text segment**: the code comprising a compiled program goes in the text segment.
- **Globals**: the compiler allocates space for any global variables
- **Heap**: when things are allocated via `new`, they come from the heap
- **Stack**: stack frames are created on the stack, which grows downward
- **Big void**: since we don't know howbig either of these will get, we keep a big hole in between the two

– **Little void**: the first thousand addresses starting at zero are reserved for accidental use of as an `int` as a pointer, resulting in a SEGFAULT.

15.6 Global vs. Local vs. Dynamic

	Global	Local	Dynamic
Where in code?	Outside function	Inside function (block) or args	Anywhere you use new
When created?	Beginning of program	Beginning of function (block)	You call new
When destroyed?	End of program	End of function	You call delete
Size?	static	static	dynamic
Location?	Globals	Stack	Heap

– See example from lecture slides.

15.7 Classes and dynamic memory

– When you create instances of classes, their constructors are called, just as if it were created the "normal" way.

```
1 IntSet *isp = new IntSet;
```

1. Allocate enough space on the heap to hold an `IntSet`
2. Call the constructor `IntSet::IntSet()` on this new object

16 Copying Arrays

16.1 Default Argument

– One way to solve this problem of duplicate definitions is to use something called a *default argument*

– **Default argument**: make an argument optional by specifying the value to be assigned when not specified (default)

```
1 IntSet(int capacity = 0);
```

16.2 Destructor

– Problem: what happens if we have a local `IntSet` inside of a function and the function returns?

```
1 void foo() {
2     IntSet is(3);
3 }  // foo returns
4
5 int main() {
6     foo();
7 }
```

– Array is stored on the heap, and only the pointer will be destroyed.

- To solve this memory leak, we have to arrange to de-allocate the integer array whenever the "enclosing" IntSetis destroyed.
- We do this with a **destructor**, the opposite of a constructor
- A destructor also has the same name as the class, preceded with a tilde
- e.g.,

```
class IntSet{
public:
    //...

    //EFFECTS: destroys this IntSet
    ~IntSet();

    //...
};

IntSet::~IntSet() {
    delete[] elts;
}
```

- Destructors run automatically when an object is destroyed
- When does the destructor run?
  - Local: out of scope
  - Global: program end
  - Dynamic: **when it is deletec**

16.3 Copy Constructor

- In our original `IntSet` when a copy is currently performed the pointer to the array of the original set is copied, so both sets will have pointers point to one array.
- The copy constructor has two tasks
  1. Initialize the member variables
  2. **Copy everything** from the other `IntSet`
- e.g.,

```
IntSet::IntSet(constIntSet&other) {
    //1. Initialize member variables
    elts= new int[other.elts_capacity];
    elts_size= other.elts_size;
    elts_capacity= other.elts_capacity;

    //2. Copy everything from the other IntSet
```

```
8       for (inti= 0; i< other.elts_size; ++i)
9           elts[i] = other.elts[i];
10  }
```

- The copy constructor we've written follows pointers and copiesthe things they point to, rather than just copying the pointers
- This is called a **deep copy**, as opposed to the default behavior which called a **shallow copy**

16.4 Destructors and polymorphism

- When you create a object, all the constructors run, starting with the base class
- Now, let's see what happens when we mix destructors with polymorphism
- e.g.,

```
1  class Animal {
2      virtual void talk() {}
3      virtual ~Animal {}
4  };
5
6  class Chicken : public Animal {
7  public:
8      virtual void talk()
9          { cout<< "cluck\n"; }
10     virtual ~Chicken {}
11 };
12
13 class Horse : public Animal {
14 public:
15     virtual void talk()
16         { cout<<  neigh\n"; }
17 };
```

- When running the following code:

```
1  Animal *a = ask_user(); //input:  Chicken
2  // do something with a
3  delete a; a=0;
```

- Since dtoris virtual, correct dtors(`~Chicken()`, then `~Animal()`) are selected at runtime.
- Destructors run from child class towards parent class!

17 Deep Copies and Resizing

17.1 Problems with assignment "="

- Same issue with copy constructor, it does a shallow copy

- **Operator overloading** lets us customize what happens when we use a built-in symbol.

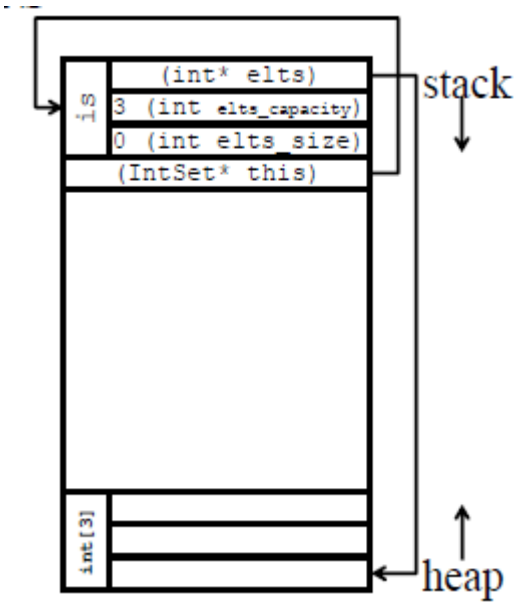
```
class IntSet{
    // data elements
    // ...

public:
    // Constructors
    // EFFECTS: assignment operator does a deep copy
    IntSet &operator= (const IntSet &rhs);
    //...
};
```

- Like the copy constructor, the assignment operator takes a reference to a const instance to copy from
- However, it also returnsa reference to the copied-to object.
- This return value allows for a chained assignmentlike this: `is3= is2= is1;`

17.2 Understanding this

- We need to return a reference to the current `IntSet` so that chaining works
- Every member function has a "secret" variable called `this`
  - `this` is a pointer to the current instance of the class
  - Here, we use `*this` because the reutrn type isa reference
- Think of `this` as a local variable which points to the current instance



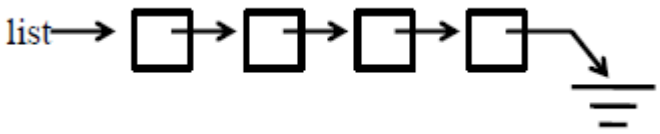
17.3 The Rule of the Big Three

– If you have any dynamically allocated storage in a class, you must provide:

1. A destructor
2. A copy constructor
3. An overloaded assignment operator

18 Linked Lists

– A linked structure is one with a series of zero or more pieces of data, connected by pointers from one to another



– A list is another example of a container ADT

18.1 List nodes

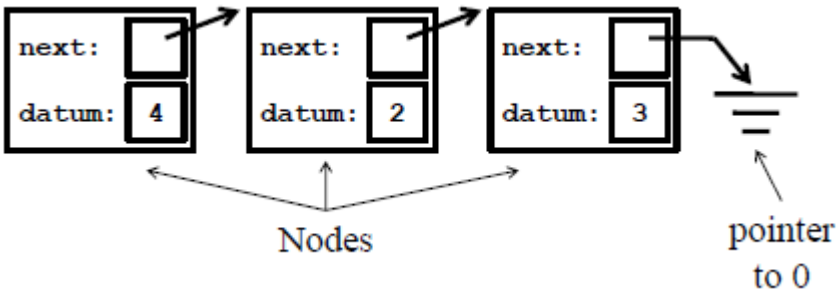
– We need to pick a concrete representation that stores a list as a dynamically created list of "nodes"

```
1 structNode {
2     Node *next;
3     int datum;
4 };
```

– Invariant:

  – the datum field holds the integer datum of an element in the list
  – Invariant: the next field points to the next Nodein the list, or 0 (AKA NULL) if no such Node exists

– Resulting in the list's concrete implementation in the form of:



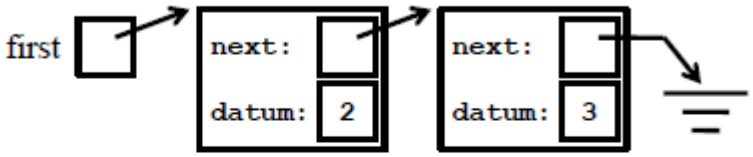
```
1 class IntList{
2     //...
3 private:
4     struct Node {
5         Node *next;
6         int datum;
7     };
8     Node *first;
9 };
```

– Representation invariant: `first` points to the first node in the sequence of nodes representing this `IntList`, or 0 if the list is empty

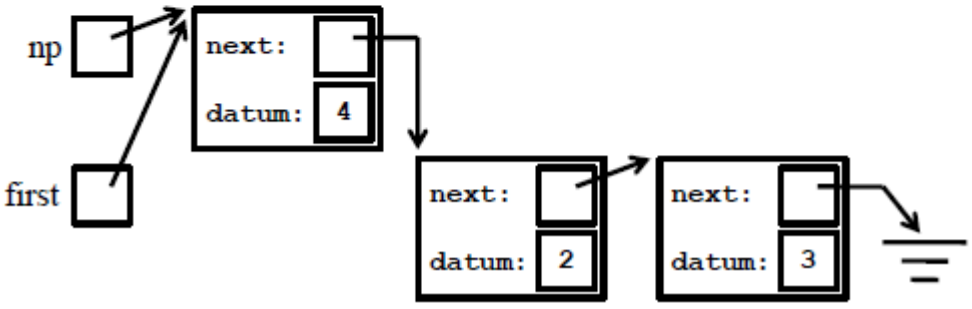
18.2 Insert

```
1 void IntList::insertFront(int i) {
2     Node *np= new Node;
3     np->datum = i;
4     np->next = first;
5     first = np;
6 }
```

- Suppose we are inserting a 4. The list might already have elements:



- And then the list's invariant :



18.3 Implementing removeFront()

– If we are removing this first node, we must delete it to avoid a memory leak

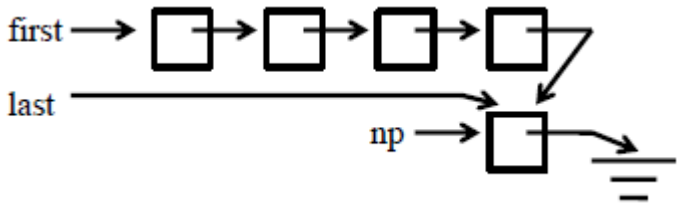
– Unfortunately, we can't delete it before advancing the first pointer

```
int IntList::removeFront() {
    assert(!isEmpty());
    Node *victim = first;
    first = first->next;
    int result = victim->datum;
    delete victim; victim=0;
    return result;
}
```

18.4 Implementing insertBack()

– Adding a member variable to point to the last item in a list, allows for trivial insert back

```
void IntList::insertBack(inti) {
    Node *np= new Node;
    np->datum = i;
    np->next = 0;
    if (isEmpty()) {
        first = last = np;
    } else {
        last->next = np;
        last = np;
    }
}
```



– Removing from back is expensive because you must find the previous to last node: $\mathcal{O}(n)$

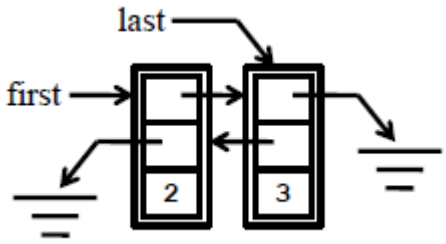
18.5 Singly linked vs. doubly linked lists

– To make removal from the end efficient, as well, we have to have a doubly-linked list, so we can go forward and backward.

– In our new representation, a node is:

```
structNode {
    Node *next;
    Node *prev;
    int datum;
}
```

– e.g., [2, 3]



19 Templated Containers

19.1 Introduction

– Abstract Data Types like `IntSet` and `IntList` are often called **containers** or container classes, since their purpose is to contain other objects

– Today, we will use the C++ `template` mechanism to reuse the same container code for any type

– To see how templates might be useful, consider the following fragments defining a list-of-intand a list-of-char:

```
1  // Int
2  class List {
3  public:
4      void insertFront(int v);
5      int removeFront();
6      //...
7  private:
8      struct Node {
9          Node *next;
10         int datum;
11     };
12     Node *first;
13 };
14
15 // Char
16 class List {
17 public:
18     void insertFront(char v);
19     char removeFront();
20     //...
21 private:
22     struct Node {
23         Node *next;
24         int datum;
25     };
26     char *first;
27 };
```

– It's like someone took the list-of-int definition and replaced each instance of `int` with an instance of `char`.

19.2 Templates

– The intuition behind templates is that they are code with the "type name" left as a (compile-time) constant.

– The parameter is a type, not a variable.

– To start, you first need to declare that something will be a template:

```
1 template <typename T>
2 class List {
3     // ...
4 };
```

– `T` stands for "the name of the type contained by this `List`"

– Alternate notation:

```
1 template <class T>
2 class List {
3     // ...
4 };
```

– e.g.,

```
1  template <typename T>
2  class List {
3  public:
4      // methods
5
6      // constructors/destructor
7
8  private:
9      struct Node {
10         Node *next;
11         T datum;
12     };
13     //...
14 };
```

19.3 Templated Methods

– All that is left is to define each of the method bodies.

– Each body must also be declared as a templated method and we do that in much the same way as we do for the class definition

– Each function begins with the template declaration: `template <typename T>`

– And each method name must in the `List<T>` scope:

```
template <typename T>
void List<T>::insertFront(Ti) {
    Node *np= new Node;
    np->datum = i;
    np->next = first;
    first = np;
}
```

19.4 Using Templated Containers

- To use the templated container, you specify the type `T` when creating the container object.

```
// Create a list of integers on the stack
List<int> li;
li.insertFront(42);

// Create a list of integers on the heap
List<int> *lip = new List<int>;
lip->insertFront(42);
```

- All templated codes goes in the header file: both the declaration and definition

19.5 Containers of large values

- Copying elements by value is fine for types with small representations, for example, built-in types like `int`
- This is nottrue for larger types any nontrivial structor class would be expensive to pass by value, because you'll spend all of your time copying
- e.g.,

```
class Gorilla {
    // OVERVIEW: a big, expensive class
    // lots of member data ...
};
```

- Two options:
  - Pass-by-pointer (results in container of pointers)
  - Pass-by-reference (only fixes parameter copy issues)

Container of pointers

- The good news: `Gorilla` is passed by pointer, no copies!
- The bad news: containers-of-pointers are subject to two broad kinds of potential bugs:
  1. Using an object after it has been deleted

```
List<Gorilla*> zoo;
Gorilla *colo= new Gorilla;
zoo.insertFront(colo);
// ...
delete colo; // bad!
// ...
Gorilla *g = zoo.removeFront();//g already deleted!
```

2. Leaving an object orphaned by neverdeleting it

19.6 Pattern of use for container-of-ptr

1. **Existence**: Allocate an object before inserting it into any container.

```
// OK
List<Gorilla*> zoo;
Gorilla *colo= new Gorilla;
zoo.insertFront(colo);

// better
List<Gorilla*> zoo;
zoo.insertFront(new Gorilla);
```

2. **Ownership**: Once it is inserted, do not modify it until it is removed.

```
List<Gorilla*> zoo;
Gorilla *colo= new Gorilla;
zoo.insertFront(colo);
colo->set_name("Bill"); // bad! - could mess up a container's representation
   invariant

// Good:
Gorilla *g = zoo.removeFront(); //remove
g->set_name("Bill"); //change
zoo.insertFront(g); //replace
```

3. **Conservation**: Once it is removed, it must either be deallocatedor inserted into some container. Dont forget to delete at the end!

```
// bad example
List<Gorilla*> zoo;
zoo.insertFront(new Gorilla);
zoo.removeFront(); //bad! memory leak!

// good example
List<Gorilla*> zoo;
zoo.insertFront(new Gorilla);
deletezoo.removeFront(); // fixed
```

19.7 Deleting a shared object

- How to delete objects that occur on multiple containers?
- One solution:
  - Remove object from allcontainers before freeing up the object.

20 Iterators

- An iterator allows you to traverse a container
- Weve seen something like this before with arrays

```
int a[SIZE]; // fill a
for (int i=0; i< SIZE; ++i)
    cout<< a[i] << endl;
```

- How would you do this for a linked list, like our `List` type?
- In the end, our iterator will work a lot like using a pointer to traverse an array

20.1 Iterator functions

1. **Create**: an iterator with a constructor
2. **Get**: the `T` at the iterators current position
3. **Move**: the iterator to the next position
4. **Compare**: two iterators

```
<template T>
class Iterator {
    Node* node_ptr;
public:
    Iterator();                              // Create
    T& operator* () const;                   // Get
    Iterator& operator++ ();                 // Move
    bool operator!= (Iterator rhs) const;    // Compare
};
```

20.2 Friends

- Our list also needs to functions `begin(), end()`
- `begin()`: returns an iterator object pointing to first list position
- `end()`: returns a default iterator object "past the end" position
- Our `Iterator` needs access to the internal workings of the `List` in order to perform these functions

- We can solve this by making `Iterator` a **friend** of `List`
- Friendship is **given**, not taken
- The class declares which classes are its friends
- e.g.,

```
class List {
    // ...
    class Iterator {
        // ...
        friend class List;
    };
};
```

20.3 Using iterators

- With these functionalities we can now use an iterator to traverse a list:

```
List<int> l;
l.insertFront( 3 );
l.insertFront( 2 );
l.insertFront( 1 );

for (List<int>::Iterator i=l.begin(); i!= l.end(); ++i) {
    cout<< *i<< " ";
}

cout<< endl;
```

- We have successfully created a function abstraction to access a container

20.4 Iterator invalidation

- Once an iterator is created, if the underlying container is modified, the iterator may become invalid.
- If the iterator is invalidated, its behavior is undefined, much like an invalid pointer.
- The intuition behind this is that the iterator depends on the representation of the container if that changes, the iterator is likely to miss an element or return an element that no longer exists.
- e.g., (BAD)

```
List<int> l;
l.insertFront( 1 );
List<int>::Iterator i=l.begin();
++i; ++i; // oops, went off the end!
```

- Its your responsibility to keep your Iterator within the bounds of the container

20.5 Putting it all together

```
List<Gorilla*> zoo;
zoo.insertFront(new Gorilla("Colo"));
zoo.insertFront(new Gorilla("Koko"));

for (List<Gorilla*>::Iterator i=zoo.begin(); i!= zoo.end(); ++i)
    cout<< (**i).get_name() << endl;

for (List<Gorilla*>::Iterator i=zoo.begin(); i!= zoo.end(); ++i) {
    delete *i; *i=0;
}

cout<< endl;
```

21 Functors

21.1 First-class Objects

- For any entity in a programming language, we say that the entity is **first-class** if you can do all four of these things:
  1. Create them
  2. Destroy them
  3. Pass them as arguments
  4. Return them as values
- Unlike values, types and functions are notfirst class objects in C++; but, they can sometimes come close.
- The **template mechanism** allows us to pass types as arguments
- The **function pointer mechanism** allows us to pass functions as arguments and return them as results

21.2 Motivation

- `any_even, any_odd` are both very similar functions.
- We would like a function that is generic, and could perform a more general form of this operation
- The generic function signature would like like this:

```
bool any_of(const List<int> &l, bool (*pred)(int));
```

21.3 Using function pointers

- `pred` is a pointer to a function that takes a single integer argument, returning a boolean result
- A **predicate** is used to control an algorithm
- The generic implementation is:

```
bool any_of(const List<int> &l, bool (*pred)(int)) {
    for(List<int>::Iterator i=l.begin(); i!=l.end(); ++i)
        if (pred(*i)) return true;

    //reached end without finding elt
    return false;
}
```

– We can get the odd elements, by using the appropriate predicate:

```
bool isOdd(intn) {
    return (n%2);
}

List<int> l;
// fill l ...
bool contains_odd= any_of(l, isOdd);
```

21.4 Functors

– We can redefine the paranthesis operator () and use a class as a function

– However, unlike functions with function pointers, objects can have per-object state, which allows us to specialize on a per-object basis.

– e.g.,

```
class GreaterN{
    int limit;
public:
    GreaterN(int limit_in) : limit(limit_in) {}
    bool operator() (int n) {
        return n > limit;
    }
};
```

– We can use this class to "generate" specialized functors:

```
GreaterNg2(2);
GreaterNg6(6);
cout<< g2(4) << endl;
cout<< g6(4) << endl;
```

– Another use for functors: **Comparison**: used to define order, takes two elements of the same type and ouputs a `bool`

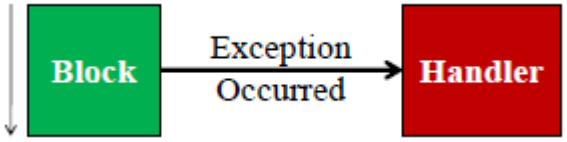
22 Exceptions

22.1 Detecting errors at runtime

- We want a means of recognizing and handling unusual conditions in your program at runtime, not just at compile time.
- A REQUIRES clause is just comments and cannot enforce the specification. Therefore, it is easy to pass parameters that violate the specification.

22.2 Exceptions

- **Exceptions** let us detect an error in one part of the program and correct it in a different part of the program
- When an exception occurs, control transfers from normal-case code to the error handling code
- This error handling code then tries to correct the problem



- When an exception occurs, it propagates from a function to its caller until it reaches a handler
- This is called **exception propagation**, and it happens automatically

22.3 Exception Handling in C++

- When code detects an error, it uses a `throw` statement
- Code that might cause an error goes in a `try` block
- Code that corrects an error goes in a `catch` block
- If the exception is successfully handled in the catch block, execution continues normally with the first statement following the catch block
- Otherwise, the exception is propagated to the enclosing block or to the caller if there is no enclosing block
- If an exception is propagated to the caller of `main()`, the program exits
- When code detects an error, it uses a `throw` statement
- When we throw an exception, we specify a value for the exception type in a throw statement
- e.g.,

```
1 int n = 0; throw n;
2 char c = 'e'; throw c;
```

22.4 Terminology

- Throw exception == raise exception
- Catch block == exception handler

22.5 Exception Example

- Code that might cause an error goes in a `try` block:

```
int main() {
    string f; //function
    double n; //number

    while (cin>> f >> n) {
        try {
            if (f == "factorial") {
                cout<< factorial(n) << endl;
            }
        }catch(int i){
            cout << "try again" << endl;
        }
    }
}
```

- Code that corrects an error goes in a `catch` block
- A `catch` block goes directly after a `try` block
- A `catch` block matches the type from a `throw` statement

```
./a.out
factorial 5
120
factorial -5
try again
```

- When an exception is not caught by a catch block, it propagates all the way to the caller of `main`, and the program exits

```
./a.out
combination -5 4
\item terminate called after thrown an instance of 'int'
Aborted (core dumped)
```

22.6 Type discrimination

- A `try` block can have multiple `catch` blocks to handle different exception types

```
try {
    if (foo) throw 4;
    // some statements go here
    if (bar) throw 2.0;
    // more statements go here
    if (baz) throw    a   ;
}
catch (intn) { }
catch (double d) { }
catch (char c) { }
catch (...) { }
```

- The last handler is a **default handler**, which matches any exception type. It can be used as a "catch-all" in case no other catch block matches.

22.7 Exception types

- Code often uses custom types to describe errors:
- e.g.,

```
class NegativeError{};
class InputError{};
```

- We use the class mechanism to declare custom types
- When an error is detected, create a `NegativeError` object and `throw` it:

```
//EFFECTS: returns n!, throws NegativeErrorfor n<0
int factorial (int n) {
    if (n<0) throw NegativeError();
    int result = 1;
    while (n > 0) {
        result *= n;
        n -= 1;
    }
    return result;
}

int main() {
    //...
    while (cin >> f >> n) {
        try {
            //...
        } catch (NegativeError){
            cout<< "try a positive number" << endl;
        } catch (...) {
            cout<< "try again" <<endl;
        }
```

```
22         }
23 }
```

– See examples from lecture slides.