Lecture 22

Final Review

Shibo Li

shiboli@cs.fsu.edu

Department of Computer Science
Florida State University

Part I: C++

Basic Creation and Execution of a C++ program

FSU

- Create **source code** with a text editor
  - Source code is just a plain text file, usually given a filename extension to identify the programming language (like .c for C, or .cpp for C++)
- **Preprocessor** - Part of compiler process, performs any pre-processing tasks on source code.
- **Compilation** - syntax checking, creation of object code. Object code is the machine code translation of the source code.
- **Linking** - Final stage of the creation of an executable program. Linking of object code files together with any necessary libraries (also already compiled).
- **Execution of program** - Program loaded into memory, usually RAM, CPU executes code instructions.

Data Types

Primitive/Fundamental data types: are the built-in types defined by the C++ language. These types represent the most basic forms of data that the language can manipulate directly, without the need for any additional libraries or user-defined classes. (see `data_types.cpp`)

- **bool:** has two possible values, true or false
- **char:** represents a single character.
  - Typically 1 byte
  - Stored with an integer code underneath (ASCII on most computers today)

Data Types

- **integer:** has two possible values, true or false
  - **short** - (usually at least 2 bytes)
  - **int** - (4 bytes on most systems)
  - **long** - (usually 4 or more bytes)
  - The integer types have regular and unsigned versions
- **floating point types:** for storage of decimal numbers (i.e. a fractional part after the decimal)
  - **short** - 4 bytes
  - **double** - 8 bytes
  - **long double** - more than 8 bytes

Initializing Variables

- To **declare** a variable is to tell the compiler it exists, and toreserve memory for it
- To **initialize** a variable is to load a value into it for the first time
- If a variable has not been initialized, it contains whatever bits are already in memory at the variable's location (i.e. a garbage value) — This is a very common mistake and hard to debug. *(see code example var_init.cpp)*

Constants

- A variable can be declared to be **constant.** This means it **cannot change once it's declared and initialized.**
- Use the keyword `const`
- **MUST** declare and initialize on the same line *see const_test.cpp*

```
const int SIZE = 10;
const double PI = 3.1415;
// this one is illegal, because it is not
// initialized on the same line
const int LIMIT; // BAD!!!
LIMIT = 20;
```

- A common convention is to name constants with **all-caps** (not required)

Symbolic Constants (An Alternative)

- A symbolic constant is created with a preprocessor directive, `#define`. (This directive is also used to create macros).
- Examples:

```
#define PI 3.14159
#define DOLLAR '$'
#define MAXSTUDENTS 100
```

- The preprocessor replaces all occurrences of the symbol in code with the value following it. (like find/replace in MS Word).
- This happens before the actual compilation stage begins.

const vs #define

- **Type Safety:**
  - `const`: has a specific type, which is checked by the compiler
  - `#define`: no specific type, simply text substitutions
- **Scope:**
  - `const`: subject to C++ scoping rules
  - `#define`: globally visible from the point of definition

Input and Output Streams

- In C++ we use do I/O with "stream objects", which are tiedto various input/output devices.
- These stream objects are predefined in the iostream library.
- **cout** - standard output stream
  - Of class type `ostream` (to be discussed later)
  - Usually defaults to the monitor

Input and Output Streams

- **cin** - standard input stream
  - Of class type `istream` (to be discussed later)
  - Usually defaults to the keyboard
- **cerr** - standard error stream
  - Of class type `ostream`
  - Usually defaults to the monitor, but allows error messages to be directed elsewhere (like a log file) than normal output

Operators

- Special built-in symbols that have functionality, and work on operands
- **operand** - an input to an operator
- **Arity** - how many operands an operator takes
  - *unary operator* - has one operand
  - *binary operator* - has two operands
  - *ternary operator* - has three operands
- Examples:

```
int x, y = 5, z;
z = 10; // assignment operator (binary)
x = y + z; // addition (binary operator)
x = -y; // -y is a unary operation (negation)
x++;  // unary (increment)
```

- **cascading** - linking of multiple operators, especially of related categories, together in a single statement:

```
int x, y = 5, z;
// cascading arithmetic operators
x = a + b + c - d + e;
// cascading assignment operators
x = y = z = 3;
```

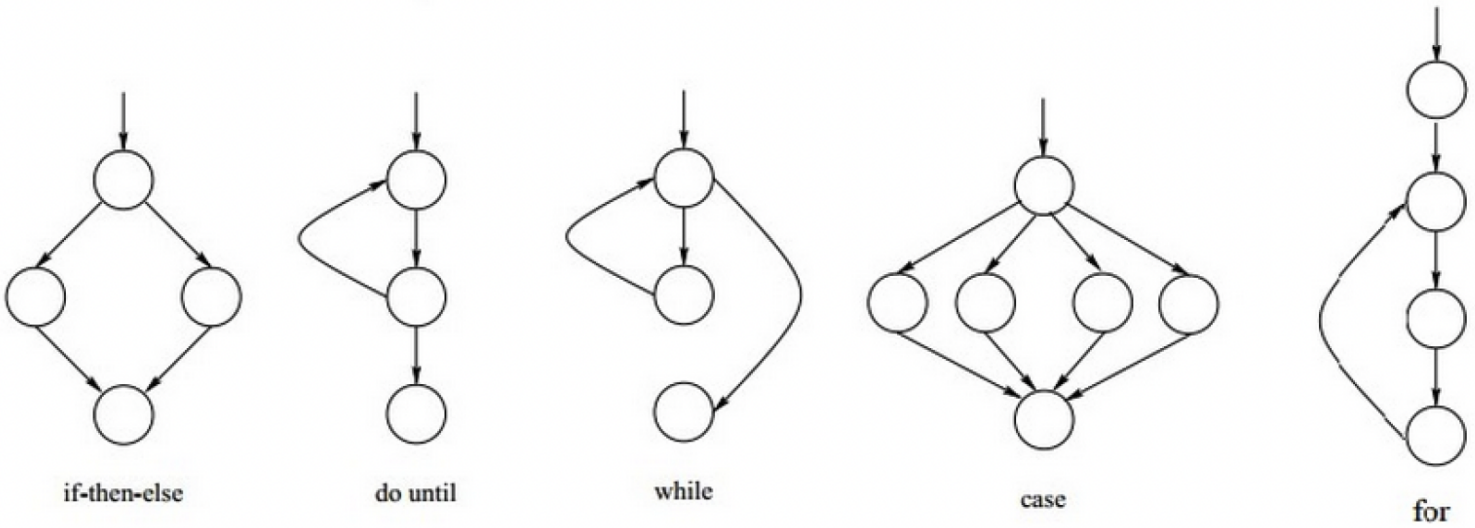
- **Precedence** - rules specifying which operators come first in a statement containing multiple operators

```
x = a + b * c; // b * c happens first, since *
                  // has higher precedence
                     than +
```

- **Associativity** - rules specifying which operators are evaluated first when they have the same level of precedence.
  - Most (but not all) operators associate from left to right.

Control Flow

Control flow refers to the specification of the order in which the individual statements, instructions or function calls of an imperative program are executed or evaluated



Types of Control Flow

Flow of control through any given function is implemented with three basic types of control structures:

- **Sequential:** Default mode. Statements are executed line by line.
- **Selection:** Used for decisions, branching – choosing between 2 or more alternative paths.
  - `if`
  - `if-else`
  - `switch`
  - other conditional states

Types of Control Flow

- **Repetition:** Used for looping – repeating a piece of code multiple times in a row.
  - `while`
  - `do-while`
  - `for`
- The function construct, itself, forms another way to affect flow of control through a whole program. This will be discussed later in the course.

Relational Operators

Relational Operators are use for comparison.

The comparison operators in C++ work much like the symbols we use in mathematics. Each of these operators returns a Boolean value: a true or a false.

```
x == y // x is equal to y
x != y // x is not equal to y
x <y // x is less than y
x <= y // x is less than or equal to y
x >y // x is greater than y
x >= y // x is greater than or equal to y
```

Logical Operators

C++ has operators for combining expressions. Each of these operators returns a boolean value: a true or a false.

```
!x // the NOT operator (negation) true if x is false
x && y // the AND operator true if both x and y are
    true
x || y // the OR operator true if either x or y or
    both are true
```

The Switch Statement

A switch statement is often convenient for occasions in which there are multiple cases to choose from. The syntax format is:

```
switch (expression)
{
   case constant:
      statements
   case constant:
      statements

   ...(as many case labels as needed)

   default: // optional label
      statements
}
```

The Conditional Operator

There is a special operator known as the conditional operator that can be used to create short expressions that work like `if/else` statements.

```
test expr ? true expr : false expr
```

- The test expression is evaluated for true/false value. This is like the test expression of an if-statement.
- If the expression is true, the operator returns the true expression value.
- If the test expression is false, the operator returns the false expression value.
- Note that this operator takes three operands. It is a ternary operator in the C++ language

Conditional Operator Examples

```
cout <<(x >y) ? "x is greater than y" : "x is less
    than or equal to y");
// Note that this expression gives the same result
    as the following
if (x >y)
   cout <<"x is greater than y";
else
   cout <<"x is less than or equal to y");
```

Repetition Statements

- Repetition statements are called loops, and are used to repeat the same code mulitple times in succession.
- The number of repetitions is based on criteria defined in theloop structure, usually a true/false expression
- The three loop structures in C++ are:
  - `while`
  - `do-while`
  - `for`
- Three types of loops are not actually needed, but having the different forms is convenient

while and do-while loops

- Format of `while` loop:

```
while (expression)
   statement
```

- Format of `do/while` loop:

```
do
   statement
while (expression);
```

- The expression in these formats is handled the same as in the `if/else` statements discussed previously (0 means false, anything else means true)
- The "statement" portion is also as in if/else. It can be a single statement or a compound statement (a block {}).

The `for` loop

- The `for` loop is most convenient with counting loops – i.e., loops that are based on a **counting variable, usually a known number of iterations**
- Syntax of `for` loop

```
for (initialCondition; boolean Expression;
    iterativeStatement)
```

break and continue

- These statements can be used to alter the flow of control in oops, although they are not specifically needed. (Any loop can be made to exit by writing an appropriate test expression).
- break: This causes immediate exit from any loop (as well asfrom switch blocks).
- continues: When used in a loop, this statement causes the current loop iteration to end, but the loop then moves on to the next step.
  - In a while or do-while loop, the rest of the loop body is skipped, and execution moves on to the test condition.
  - In a for loop, the rest of the loop body is skipped, and execution moves on to the iterative statement.

Functions

- A function is a reusable portion of a program, sometimes called *procedure* or *subroutine*.
  - Like a mini-program (or subprogram) in its own right
  - Can take in special inputs (arguments)
  - Can produce an answer value (return value)
  - Similar to the idea of a function in mathematics

- With functions, there are 2 major points of view.
  - **Builder** of the function – responsible for creating the declaration and the definition of the function (i.e., how it works)
  - **Caller** – somebody (i.e. some portion of code) that uses the function to perform a task

Why write and use functions?

- **Divide-and-conquer**
  - Can breaking up programs and algorithms into smaller, more manageable pieces
  - This makes for easier writing, testing, and debugging
  - Also easier to break up the work for team development
- **Reusability**
  - Functions can be called to do their tasks anywhere in a program, as many times as needed
  - Avoids repetition of code in a program
  - Functions can be placed into libraries to be used by more than one "program"

Function Overloading

The term **function overloading** refers to the way C++ allows more than one function in the same scope to share the same name—as long as they have **different parameter lists**

- The rationale is that the compiler must be able to look at any function call and decide exactly which function is being invoked
- Overloading allows intuitive function names to be used inmultiple contexts

Function Overloading

- The parameter list can differ in number of parameters, ortypes of parameters, or both
- **Name mangling/Name Decoration**. The basic idea is that the compiler encodes the function's name along with its parameter types into a unique name, making it distinct from other overloaded functions.

```
int Process(double num); // function 1
int Process(char letter); // function 2
int Process(double num, int position); // function 3
```

Default Parameters

Allows a function to have default values for parameters

- Specify default values in function declaration.
- Rules
  - Default values must be provided from right to left.
  - Once a parameter has a default value, all subsequent parameters must have defaults.

```
void display(int x, int y = 10, int z = 20) {
   cout << x << " " << y << " " << z << endl;
}

display(1);       // Output: 1 10 20
display(1, 2);    // Output: 1 2 20
```

Reference Variables

- A reference is an alias(nickname) for another variable. It is created using the & symbol.
  - Must be initialized at the time of declaration.
  - No separate memory is allocated for references.

Why Reference?

- Avoids copying large structures
  - Passing function parameters by reference to avoid unnecessary copies.
- Allows modification of variables passed to a function.
  - Two variables are in different scopes (this means functions)!

Comparing: Value vs. Reference

- Pass By Value
  - The local parameters are copies of the original argumentspassed in
  - Changes made in the function to these variables do not affect originals
- Pass By Reference
  - The local parameters are references to the storage locations ofthe original arguments passed in.
  - Changes to these variables in the function will affect the originals
  - No copy is made, so overhead of copying (time, storage) is saved

What is Recursion

- A recursive function is a function that calls **itself** in order to solve a smaller instance of the same problem.
- A problem is divided into smaller **sub-problems** until it reaches a **base case**, which is directly solvable.

Components of a Recursive Function

- **Base Case:** The condition under which the recursion ends. It prevents infinite recursion.
- **Recursive Case:** The part of the function that calls itself with a modified parameter, moving the problem closer to the base case.

```
function recursiveFunction(parameters) {
   if (base case condition)
      return base_case_value;
   else
      return recursiveFunction(modified_parameters);
}
```

Declaring Arrays

An array declaration is similar to the form of a normal declaration (typeName variableName), but we add on a size:

```
typeName variableName[size];
```

This declares an array with the specified size, named `variableName`, of type `typeName`. The array is indexed from 0 to size-1. The size (in brackets) must be an integer literal or a constant variable. The compiler uses the size to determine how much space to allocate (i.e. how many bytes).

Declaring Arrays

Examples

```
int list[30]; // an array of 30 integers
char name[20]; // an array of 20 characters
double nums[50]; // an array of 50 decimals
int table[5][10]; //two dimensional array of integers
```

Initializing Arrays

- Can we do the same for arrays? Yes, for the built-in types. Simply list the array values (literals) in set notation {} after the declaration. Here are some examples:

```
int list[4] = {2, 4, 6, 8};
double numbers[3] = {3.45, 2.39, 9.1};
int table[3][2] = {{2, 5} , {3,1} , {4,9}};
char letters[5] = {'a', 'e', 'i', 'o', 'u'};
```

C-style strings

Arrays of type char are special cases.

- Since character arrays are used to store C-style strings, you can initialize a character array with a string literal (i.e., a string in double quotes), as long as you leave room for the null character in the allocated space.

```
char name[7] = "Johnny";
```

- Notice that this would be equivalent to:

```
char name[7] = {'J', 'o', 'h', 'n', 'n', 'y',
    '\0'};
```

the cctype library

This C library contains useful character testing functions, as well as the two conversion functions

String I/O:

In the special case of arrays of type char, which are used to implement c-style strings, we can use these special cases with the insertion and extraction operators:

```
char greeting[20] = "Hello, World";
cout <<greeting; // prints "Hello, World"
char lastname[20];
cin >> lastname; // reads a string into 'lastname'
// adds the null character automatically
```

- Using a char array with the insertion operator << will print the contents of the character array, up to the first null character encountered

Reading strings: get and getline

- The above `cin` example is only good for reading one word at a time. What if we want to read in a whole sentence into a string?
- There are two more member functions in class istream (in the iostream library), for reading and storing C-style strings into arrays of type char. Here are the prototypes:

```
char* get(char str[], int length, char delimiter=
    '\n');

char* getline(char str[], int length, char delimiter=
    '\n');
```

C-strings vs. string objects

- In C++ (and C), there is no built-in string type
  - Basic strings (C-strings) are implemented as arrays of typechar that are terminated with the `null` character
  - string literals (i.e., strings in double-quotes) are automatically stored this way
- Advantages of `C-strings`:
  - Compile-time allocation and determination of size. This makes them more efficient, faster run-time when using them
  - Simplest possible storage, conserves space

C-strings vs. string objects

- Disadvantages of C-strings:
  - Fixed size
  - Primitive C arrays do not track their own size, so programmerhas to be careful about boundaries
  - The C-string library functions do not protect boundaries either!
  - Less intuitive notation for such usage (library features)

`string` objects

- C++ allows the creation of `objects`, specified in `class` libraries
- Along with this comes the ability to create new versions of familiar operators
- Coupled with the notion of dynamic memory allocation (not yet studied in this course), objects can store variable amounts of information inside

`string` objects

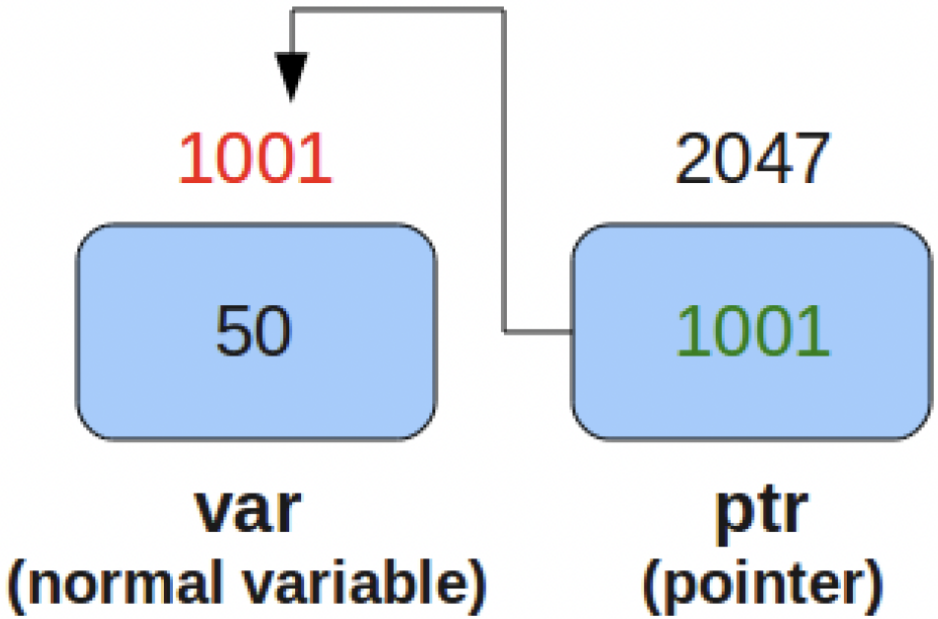
- Therefore, a string class could allow the creation of string objects so that:
  - The size of the stored string is variable and changeable
  - Boundary issues are handled inside the class library
  - More intuitive notations can be created

`string` objects

- Strings are declared as regular variables (not as arrays), and they support:
  - the assignment operator $=$
  - comparison operators $==$, $!=$, etc
  - the $+$ operator for concatenation
  - type conversions from c-strings to string objects
  - a variety of other member functions
- To use this library, make sure to #include it:`#include <string>`

What is a Pointer?

- A pointer is a variable that stores a memory address.
- Pointers are used to store the addresses of other variables or memory items.



Why we need Pointers?

- Pointers allow direct access and manipulation of memory. This is crucial in performance-critical application.
- Pointers are used to manage dynamic memory. This allows for the creation of variables or objects at runtime, which is essential when the size of data structures (like arrays) isn't known at compile time.
- When passing large objects (like structs or classes) to functions.

Declaring pointers:

- Pointer declarations use the * operator. They follow this format:

```
typeName * variableName;
int n; // declaration of a variable n
int * p; // declaration of a pointer, called p
```

- In the example above, p is a pointer, and its type will be specifically be referred to as "pointer to int", because it stores the address of an integer variable. We also can say its type is: `int*`

Declaring pointers:

- The **type** is important. While pointers are all the same size, as they just store a memory address, we have to know what kind of thing they are pointing TO.

```
double * dptr; // a pointer to a double
char * c1; // a pointer to a character
float * fptr; // a pointer to a float
```

Pointer dereferencing

- Once a pointer is declared, you can refer to the thing it points to, known as the target of the pointer, by "dereferencing the pointer". To do this, use the unary * operator:

```
cout << "The pointer is: " << ptr;
cout << "The target is: " << *ptr;
```

The null pointer

- here is a special pointer whose value is 0. It is called the null pointer
- You can assign 0 into a pointer:

```
int * ptr;
ptr = 0;
```

- The null pointer is the only integer literal that may be assigned to a pointer. You **may NOT assign arbitrary numbers to pointers**

The "address of" operator

- Recall, the & unary operator, applied to a variable, gives its address:

```
int x;
// the notation &x means "address of x"
```

- This is the best way to attach a pointer to an existing variable:

```
int * ptr; // a pointer
int num; // an integer
ptr = &num; // assign the address of num to ptr //
    now ptr points to "num"!
```

Pass by Address/Pointers

- If a pointer type is used as a function parameter type, then an actual address is being sent into the function instead
  - In this case, you are not sending the function a data value –instead, you are telling the function where to find a specific piece of data
  - Such a parameter would contain a copy of the address sent inby the caller, but not a copy of the target data
  - When addresses (pointers) are passed into functions, the function could affect actual variables existing in the scope of the caller

Reference vs. Pointers

- **Memory Representation**
- **CPU Instructions and Operations**
- **Nullability and Safety**
- **Low-Level Flexibility**

Pointers and Arrays:

- With a regular array declaration, you get a pointer for free. The name of the array acts as a pointer to the first element of the array.

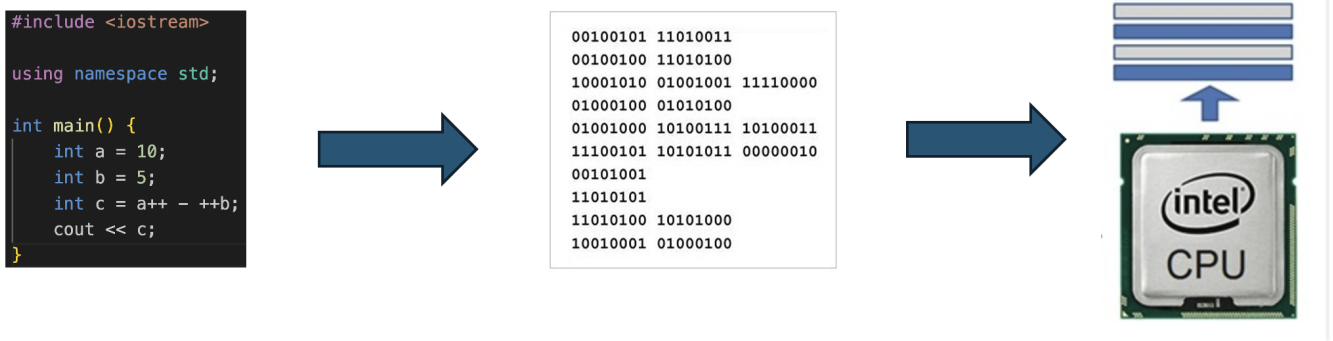
```
int list[10]; // the variable list is a pointer//
    to the first integer in the array
int * p; // p is a pointer. same type as list.
p = list; // legal assignment. Both pointers to
    ints.
```

Pointer Arithmetic

- Another useful feature of pointers is pointer arithmetic.
- When you add to a pointer, you do not add the literal number. You add that number of units, where a unit is the type being pointed to.
- Suppose ptr is a pointer to an integer, and ptr stores theaddress 1000. Then the expression (ptr + 5) does not give 1005 (1000+5).
- Instead, the pointer is moved 5 integers (ptr + (5 * size-of-an-int)). So, if we have 4-byte integers, (ptr+5) is 1020 (1000 + 5*4).

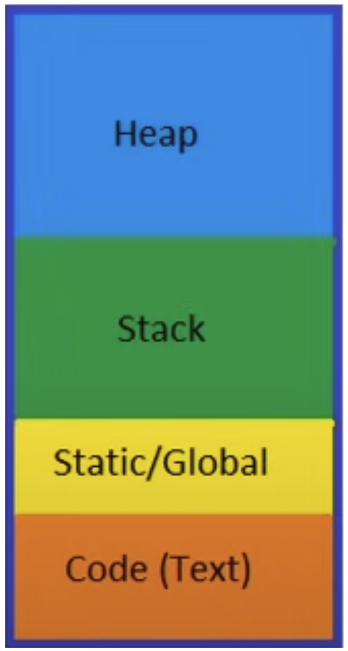
Program vs. Prcoess

- **Program:** This is a static set of instructions written in a language like C++ that is stored on disk.
- **Process:** Once a program is loaded and executed, it becomes a process. A process is an active instance of a program that runs in memory



Memory Allocation

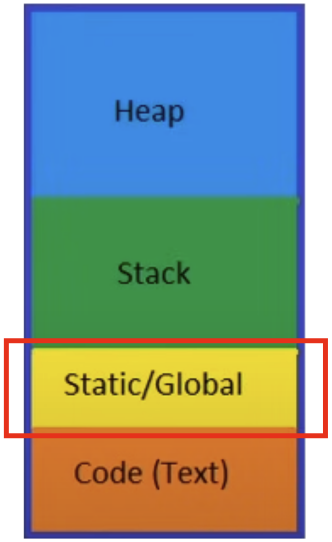
- crucial for **process** execution
- assigning specific memory areas
- data, variables, and structures



Allocating Memory

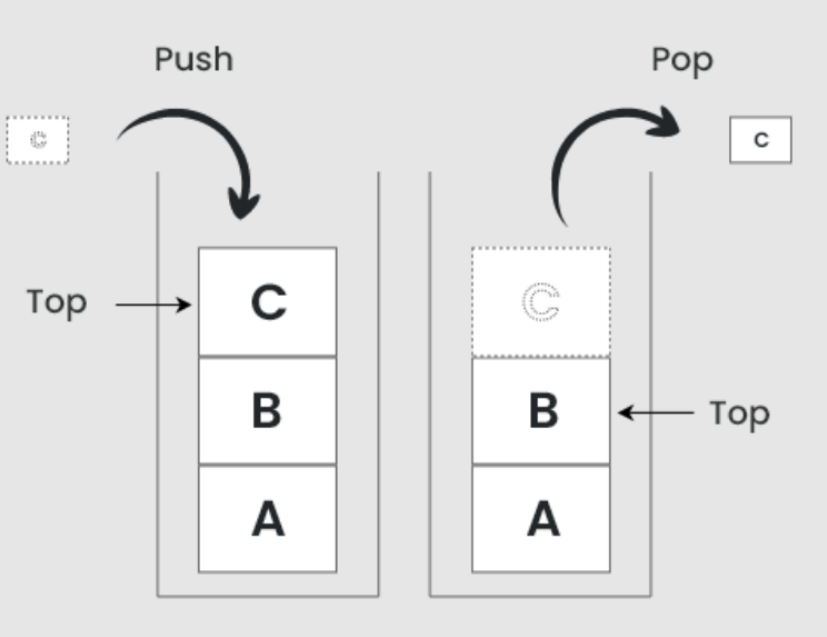
Static/Compile-Time Allocation: variables or constants before the program runs

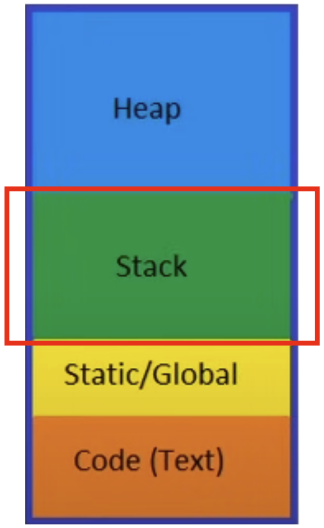
- **Global Variables**: Variables declared outside of any function are stored in the global memory segmen
- **Static Variables**: `static`
- **Constants**: `#define`, `const`



Automatic Allocation: Memory is allocated automatically on the stack

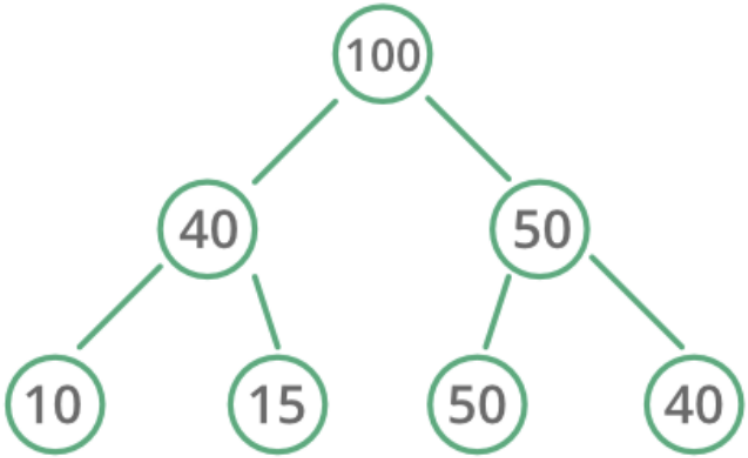
- Last in first out (LIFO)
- local variables within functions
- The stack is managed by the program's flow

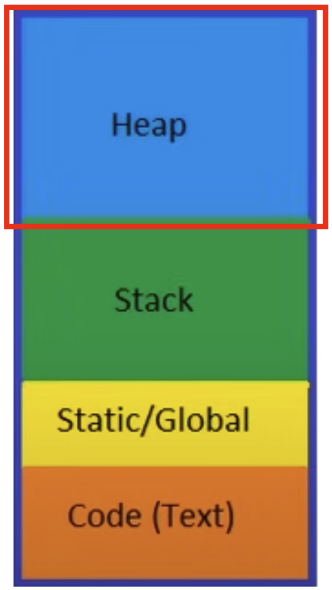




Dynamics Allocation:

- Memory allocated "on the fly" during **run time**
- Memory is managed in the heap
- Allowing for flexible memory usage but requiring explicit allocation and deallocation to avoid memory leaks





Dynamic Memory Allocation

We can dynamically allocate storage space while the programis running, but we cannot create new variable names “on the fly”

- Creating the dynamic space.
- Storing its address in a **pointer** (so that the space can be accesed)

Allocating space with `new`

- To allocate space dynamic ally, use the unary operator **new**, followed by the type being allocated.

```
new int; // dynamically allocates an int
new double; // dynamically allocates a double
```

Allocating space with new

- If creating an array dynamically, use the same form, but put brackets with a size after the type:

```
// dynamically allocates an array of 40 ints
new int[40];
// dynamically allocates an array of size doubles
// note that the size can be a variable
new double[size];
```

- These statements above are not very useful by themselves, because the allocated spaces have no names!

Dynamic De-allocation

Memory deallocation in C++ is the process of freeing up dynamically allocated memory that is no longer needed.

- Static/Stack handles deallocation automatically
- Heap handles deallocation mauanlly **by the programmer**
- If memory is not deallocated after it's no longer needed, it can lead to a **memory leak**.
- Memory that cannot be reclaimed or reused by the system, reducing the available memory for other processes

Deallocation of dynamic memory

- To deallocate memory that was created with `new`, we use the unary operator `delete`.
- The one operand should be a pointer that stores the address of the space to be deallocated:

```
int * ptr = new int; // dynamically created int
// ...
delete ptr; // deletes the space that ptr points to
ptr = nullptr;
```

- **Dangling Pointers**: Not setting pointers to nullptr after deletion can lead to accessing freed memory.

Deallocation of dynamic memory

- To deallocate memory that was created with `new`, we use the unary operator `delete`.
- The one operand should be a pointer that stores the address of the space to be deallocated:

```
int * ptr = new int; // dynamically created
    int
// ...
delete ptr; // deletes the space that ptr
    points to
ptr = nullptr;
```

- **Dangling Pointers**: Not setting pointers to nullptr after deletion can lead to accessing freed memory.

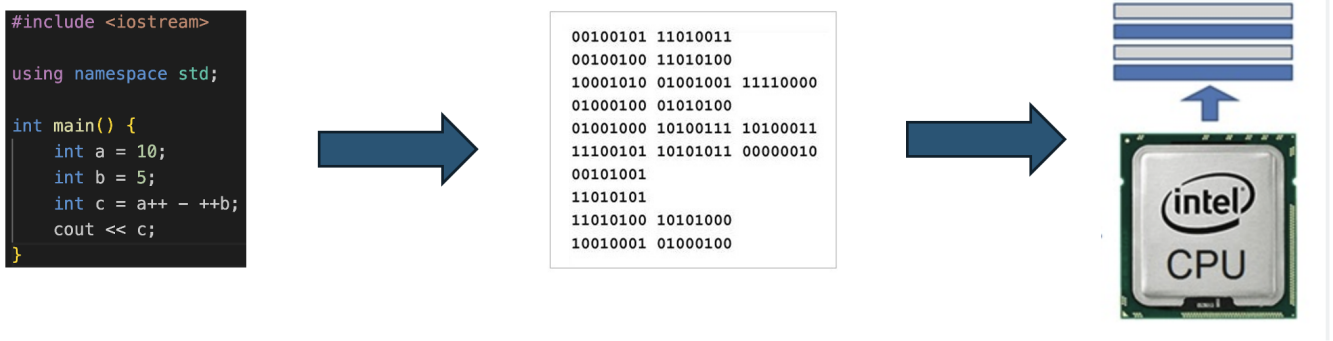
Deallocation of dynamic memory

- Note that the pointer ptr still exists in this example. That's a named variable subject to scope and extent determined at compile time. It can be reused:

```
ptr = new int[10]; // point p to a brand new array
```

Program vs. Prcoess

- **Program:** This is a static set of instructions written in a language like C++ that is stored on disk.
- **Process:** Once a program is loaded and executed, it becomes a process. A process is an active instance of a program that runs in memory



The Process Abstraction

- In the traditional systems a process executes a single sequence of instructions in an address space.
  - The program counter (PC) is a special hardware register thattracks the current instruction that is to be execute
  - In UNIX, many processes are active at the same time and the OS provides some aspects of a virtual machine
  - Processes have their own registers and memory, but rely on the OS for I/O, device control and interacting with other processes

The Process Abstraction

- Process abstraction is a key concept in operating systems that simplifies complex tasks by allowing us to view a running program as an independent, manageable entity — a process.
  - Run in a virtual address space
  - Content for resources such as processor(s), memory, andperipheral devices
  - All of the above is managed by the OS the memory management system; the I/O system; the process management and scheduling system, and the Interprocess Communication system (IPC)

The Process Abstraction

Overall, process abstraction allows developers to focus on the high-level logic of applications without needing to manage low-level system details directly.

- **Encapsulate the Execution State**: operates independently from other processes
- **Separate Resource Management**: OS takes care of scheduling, memory allocation for each process, enabling multitasking and efficient resource sharing.

The Process Abstraction

- **Facilitate Inter-Process Communication (IPC)**: Through mechanisms like pipes, message queues, and shared memory, processes can exchange information without direct access to each other's memory.
- **Ensure Isolation and Security**: Processes run in isolated memory spaces, preventing unauthorized access to each other's data, which enhances security and stability.

Multi-Processing

Multi-processing refers to a system in which multiple processes are executed **simultaneously**.

- Processes may belong to the same program or different programs
- Processes might communicate with each other via Inter-Process Communication
- Multi-processing leverages multiple CPUs or cores to execute these processes concurrently, thus improving performance in tasks that can be parallelized.

Multi-Programming

Multi-programming refers to a system where multiple programs are loaded into memory and executed concurrently by the operating system, often by quickly **switching** between them.

- Managing CPU time efficiently by keeping several programs in memory and letting them take turns running.
- System switches between programs when one is waiting for resources (like I/O), a technique called context switching.

Key Differences

- **Purpose:**
  - Executing multiple processes simultaneously
  - Efficiently managing CPU time among programs
- **Execution:**
  - Can run processes on multiple cores
  - Typically uses a single core with time-slicing

Key Differences

- **Resource Sharing:**
  - Processes are isolated, IPC needed for sharing
  - Programs share CPU and memory managed by the OS
- **Example Use:**
  - Web servers, parallel computing
  - Desktop OS running multiple applications

Why Structures

- **Compund Storage** - there are some built-in ways to encapsulate multiple pieces of data under one name
  - **Array** - we already know about this one. Indexed collections,and all items are the same type
  - **Sturcture** - keyword struct gives us another way to encapsulate multiple data items into one unit. In this case, items do not have to be the same type
- Structures are good for building records – like database records, or records in a file

What is a Structure

A structure is a collection of data elements, encapsulated into one unit.

- A structure definition is like a blueprint for the structure. It takes up no storage space itself – it just specifies what variables of this structure type will look like
- An actual structure variable is like a box with multiple data fields inside of it. Consider the idea of a student database. One student record contains multiple items of information (name, address, SSN, GPA, etc)

What is a Structure

- Properties of a structure:
  - internal elements may be of various data types
  - order of elements is arbitrary (no indexing, like with arrays)
  - Fixed size, based on the combined sizes of the internal elements

Creating Structure definitions and variables

Structure Definitions The basic format of a structure definition is:

```
struct structureName
{
   // data elements in the structure
};
```

- `struct` is a keyword
- The data elements inside are declared as normal variables. `structureName` becomes a new type.
- By themselves, these definitions above are not variables and do not take up storage

Using structures

- Once a structure variable is created, how do we use it? How do we access its internal variables (often known as its members)?
- To access the contents of a structure, we use the **dot-operator** . Format:

```
structVariableName.dataVariableName
```

Use Pointers to Structure

- But if we use parentheses to force the dereference to happen first, then it works:

```
(*fPtr).num = 10; // YES!
```

- Alternative operator for pointers: While the above example works, it's a little cumbersome to have to use the parentheses and the dereference operator all the time.
- So there is a special operator for use with pointers to structures. It is the arrow operator:

```
pointerToStruct -> dataVariable
```

Creating file stream objects, and attaching to files

- To create file stream objects, we need to include the `<fstream>` library:

```
#include <fstream> using namespace std;
```

- This library has classes `ofstream` ("output file stream") and `ifstream` ("input file stream"). Use these to declare file stream objects:

```
// create file output streams out1 and bob
ofstream out1, bob;
// create file input streams, called in1 and joe
ifstream in1, joe;
```

Creating file stream objects, and attaching to files

- File stream objects need to be attached to files before they can be used. Do this with a member function called open, which takes in the filename as an argument:

```
// For ofstreams, these calls create brand new
// files for output. For ifstreams, these calls
// try to open existings files for input
out1.open("outfile1.txt");
bob.open("clients.dat");
in1.open("infile1.txt");
joe.open("clients.dat");
```

Passing Stream Objects into Functions

- In a function prototype, any type can be used as a formal parameter type or as a return type.
  - This includes classes, which are programmer-defined types
- Streams can be passed into functions as parameters (and/or returned).
  - Because of how the stream classes were set up, they can only be passed by reference, however

Passing Stream Objects into Functions

- So, for instance, the following can be return types or parameter types in a function:

```
ostream &
istream &
ofstream &
ifstream &
```

- Why? - Functions that produce output are more flexible when they can direct their output to different destinations

Part II: Unix

Unix Commands

- `clear`
- `who`
- `whoami`
- `pwd`
- `ls`
- `cd`
- `vim`
- `g++`

Unix Commands

- `mkdir`
- `rmdir`
- `cp`
- `mv`
- `rm`
- `cd`
- `vim`
- `g++`

Unix Commands

- `date`
- `cal`
- `man`
- `du`
- `head`
- `tail`
- `sort`
- `<, >, <<, >>`

Unix Commands

- `|`
- `tar`
- `gzip`
- `diff`
- `cmp`
- `echo`
- `grep`

Shell Scripts

- A Shell Script is an executable file containing
  - Unix shell commands
  - Programming control constructs (if, then, while, until, case,for, break, continue, while, *etc.*)
  - Basic programming capabilities (assignments, variables, arguments, expressions, *etc.*)
- The file contents comprise the script

Shell Scripts

- Unlike a C++ program, that is compiled and then executed, shell scripts are **interpreted**.
- Usually, the first line of the script indicates which shell is used to interpret the script.

You First Script

```
#!/bin/sh
#this is the script in file firstscript.sh cal
date
who | grep shiboli
exit
```

- The ‘‘#!’’ is used to indicate that what follows is the shell used to interpret the script
- The ‘‘exit’’ command immediately quits the shell script (by default it will also quit at the end of the file)

Shell Script

- Variables, System Variables, Arguments
- Conditions, if, case,
- Loops, while, util, for
- exit
- expression evaluation, command substitution
- Regex and operators