C/C++ BASICS

About the notes

- This is not a comprehensive review of C/C++ Language.

- Useful online resources:
  - http://www.tutorialspoint.com/cprogramming
  - https://www.tutorialspoint.com/cplusplus/index.htm
  - https://www.geeksforgeeks.org/c-plus-plus/
  - http://www.cplusplus.com/doc/tutorial/

- References:
  - Tim Love, ANSI C for Programmers on UNIX Systems
  - Bjarne Stroustrup, The C++ Programming Language

C++ History

- Bjarne Stroustrup 1983
- Standardization:
  - First C++ standard
    - 1998 (C++98)
    - 2003 (C++03)
  - Second standard
    - 2011 (C++11)
    - 2014(C++14)
    - 2017(C++17)

C++ Standard library

- C++ standard library = C standard library + STL (standard template library)
- STL (Alexander Stepanov) provides:
  - Containers: list, vector, set, map …
  - Iterators
  - Algorithms: search, sort, …

Sec 1. Basic Facilities

- Variables
- Names
- Type Safety
- Input
- Input and type
- Operations and operators
- Assignment and initialization
- Composite assignment operators

Hello World

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

// main() function is where program execution begins.
int main()
{
   cout << "Hello World"; // prints Hello World
   return 0;
}
```

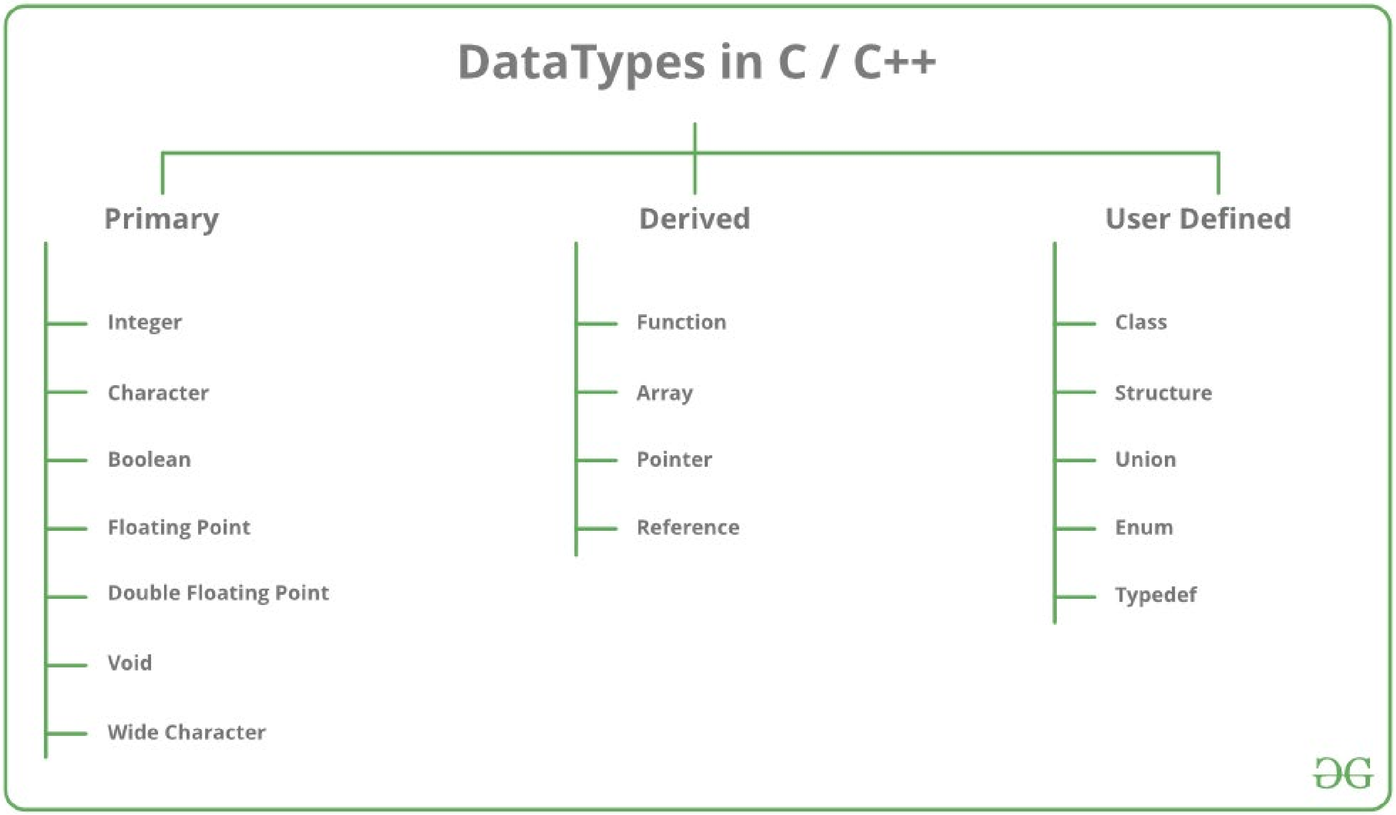
comment

- https://www.tutorialspoint.com/cplusplus/cpp_basic_syntax.htm

Variables

- Variables store data, and can do other things for us, too.
- Declare, define – to make a variable of a prescribed name
- Value – the actual value of a variable.
- Every variable has a well-defined type and is strongly typed. You cannot store incompatible types in variables.
- https://www.geeksforgeeks.org/c-data-types/

C++ variable types



- `auto` // C++11, the compiler decides the type automatically

```
auto i = 7;
```

http://www.cplusplus.com/doc/tutorial/variables/

https://www.geeksforgeeks.org/c-data-types/ (use `sizeof()` operator)

Example

```
string name = "Annamarie"; // declare and assign
int number_of_steps = 39; // declare and assign
double tolerance; // declare only, un-initialized
int I {10};          // uniform init., C++11


//incompatible types
string name2 = 39; // BAD LINE – will not compile!
int num_steps = "Annamarie"; // BAD LINE.
```

Preface – strongly statically typed

- C++ is strongly and statically typed
- Strongly typed: every variable has a type
- Statically typed: a variable has the same type for its entire lifetime.
- Other languages are not. Matlab is dynamically typed, e.g. What about R, Python?

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

// Variable declaration:
extern int a, b;
extern int c;
extern float f;
// Using global variables is
// NOT a good practice

int main ()
{
    // Variable definition:
    int a, b;
    int c;
    double df;

    // actual initialization
    a = 10;
    b = 20;
    c = a + b;
    cout << c << endl ;
    df = 70.0/3.0;
    cout << df << endl ;
    return 0;
}
```

https://www.tutorialspoint.com/cplusplus/cpp_variable_types.htm

Type Qualifiers

- `const`

Define constants

```
const int N = 10;
int t[N];
N = 20; // compilation error. Const can not be modified.
```

Protecting a parameter

```
void sayHello(const string& who){
    cout<<"Hello, "+who<<endl;
    who = "new name";  // compilation error
}
```

Legal names

- A name must start with an alphabetic character

  `` `2x` `` is not a valid name
- A name must not contain special symbols

  `` `time$2$market` `` is not a valid name
- A name cannot contain white space

  `` `milk fat` `` is not a valid name
- A name is case-sensitive

  `` `a` `` is distinct from `` `A` ``
- A name cannot be a key word. These include:
  - `if, for, while, true, string, int, double,` …

good practice for names

- Choose short-ish descriptive names
- Avoid really long names
- avoid names with mistakeable patterns.
- Do not name a variable capital eye, nor the letter oh.
- conventions exist in any language. look up the ones your employer uses.
- You can either use Google style guide (https://google.github.io/styleguide/cppguide.html) or follow Hungarian naming convention.
- See also Sakai/Resources/Appendix_1_Writing_Clean_Code.pdf

Type safety

- The type system exists to help you. It generates errors. Errors are not bad, nor is writing code which generates errors. Errors exist to help us write better code.

- Here are some type guidelines:
  - do not use a variable before giving it a value.
  - research which types are convertible.
    e.g. can you safely convert an `int` into a `double`?
    what about a `char` into an `int`?

    See https://www.tutorialspoint.com/how-do-i-convert-a-char-to-an-int-in-c-and-cplusplus

unsafe conversions

- Conversions are unsafe when they risk losing data.

- Can you convert a string to an integer?
  - https://www.geeksforgeeks.org/converting-strings-numbers-cc/
- What about a `double` into an `int`?

- C++ note: use `{}` to initialize variables, to get the compiler to help prevent unsafe conversions.

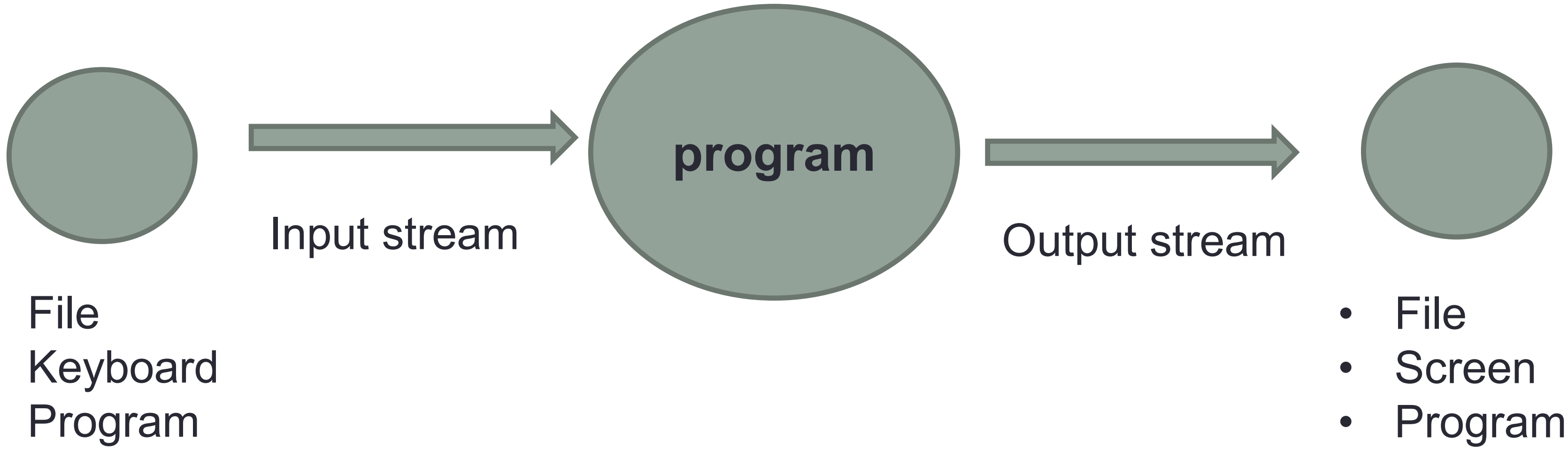
  e.g.

```
double x{2.7};   //is ok
int y{2.7};      // error!
~zxu2/Public/input_type.cpp
```

Additional reading

- https://www.tutorialspoint.com/cplusplus/index.htm
  - C++ Basics
  - …
  - C++ Storage Classes

C++ I/O streams



C++ comes with libraries which provides us many ways for performing input and output. In C++ input and output is performed in the form of sequence of bytes or more commonly known as streams.

https://www.tutorialspoint.com/cplusplus/cpp_basic_input_output.htm
https://www.geeksforgeeks.org/basic-input-output-c/

Input

- Most programs take input. Let's explore a way to get data into a program in C++.
- The main C++ object we'll use: \``std::cin`\`, which we write \``std::cin`\` or \``cin`\` depending on whether namespace ‘`std`’ is specified. \``cin`\` is an input stream, reads data from the “input console”
- The main operator we'll use: \``>>`\`, which means “get from”. Data flows in the direction of the arrows.
- e.g. \``std::cin >> obj1;`\`

Output

- The main C++ object we'll use for output: `` `std::cout` ``, which we write `` `std::cout` `` or `` `cout` `` depending on whether namespace ‘`std`’ is specified. `cout` A buffered output stream, writes data to the “output console”.
  - The main operator we'll use: `` `<<` ``, which means “insert to”. Data flows in the direction of the arrows.
-
- `std::cerr` is an unbuffered output stream, writes data to the “error console”

Input/output example

```
//read and write a first name
#include "std_lib_f.h"

using namespace std;

int main()
{
       cout << "Please enter your first name:\n";
       string first_name; // first_name is of type string
       cin >> first_name; // read characters into first_name
       cout << "Hello, " << first_name << "!\n";
}

//~zxu2/Public/src/asdf.cpp
```

Recommended way to write your own header 'std_lib_f.h'

```
#if !defined(_STD_LIB_H)
#define _STD_LIB_H

#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <new>
#include <string>

// extern double addition(double, double);

#endif
```

- 1. `//read and write a first name:` This line is a comment line. A comment is used to display additional information about the program. A comment does not contain any programming logic. When a comment is encountered by a compiler, the compiler simply skips that line of code. Any line beginning with '`//`' without quotes OR in between `/*…*/` in C++ is comment.
- .2 `#include`: In C++, all lines that start with pound (`#`) sign are called directives and are processed by preprocessor which is a program invoked by the compiler. The `#include` directive tells the compiler to include a file.
- `#include "std_lib_f.h":` tells the compiler to include the user defined file `"std_lib_f.h"`.

- `using namespace std:` This is used to import the entirety of the std namespace into the current namespace of the program. The statement "`using namespace std;`" is generally considered a bad practice. When we import a namespace we are essentially pulling all type definitions into the current scope. The std namespace is huge. The alternative to this statement is to specify the namespace to which the identifier belongs using the scope operator(`::`) each time we declare a type.

  - Namespaces allow us to group named entities that otherwise would have global scope into narrower scopes, giving them namespace scope. This allows organizing the elements of programs into different logical scopes referred to by names. (more reading: https://www.geeksforgeeks.org/namespace-in-c/)
  - See `~zxu2/Public/src`
    - A simple namespace usage example consisting of `test_name.h` and `test_namespace.cpp`.

- `int main():` This line is used to declare a function named "`main`" which returns data of integer type. A function is a group of statements that are designed to perform a specific task. Execution of every C/C++ program begins with the `main()` function, no matter where the function is located in the program. So, every C/C++ program must have a `main()` function.

- `{` and `}`: The opening braces '`{`' indicates the beginning of the main function and the closing braces '`}`' indicates the ending of the main function. Everything between these two comprises the body of the main function.

- Standard output stream (`cout`): The data needed to be displayed on output device is inserted in the standard output stream (`cout`) using the insertion operator (`<<`). Everything followed by the character "`<<`" is displayed to the output device. The output device is usually the desktop screen.

`cout << "Please enter your first name:\n"`: This line tells the compiler to display the message "`Please enter your first name:`" on the screen. This line is called a statement in C++. Every statement is meant to perform some task. A semi-colon ';' is used to end a statement. Semi-colon character at the end of statement is used to indicate that the statement is ending there.

- `string first_name;`: Declare variable `first_name` with type `string`. A variable in simple terms is a storage place which has some memory allocated to it. Basically, a variable used to store some form of data. Different types of variables require different amounts of memory, and have some specific set of operations which can be applied on them.

- Standard Input Stream (`cin`): If the direction of flow of bytes is from device(for example: Keyboard) to the main memory then this process is called input. `cin` is the instance of the class **istream** and is used to read input from the standard input device which is usually keyboard.

- The extraction operator(>>) is used along with the object `cin` for reading inputs. The extraction operator extracts the data from the object `cin` which is entered using the keyboard.

- Sometimes we can end up including a header file multiple times. C++ doesn't like this if it ends up redefining something again.

- To fix this we can use `#if !defined()` OR `#ifndef`
  - `#if !defined(_STD_LIB_H):` tells the compiler to ignore what follows if it has already seen `_STD_LIB_H` before.
  - `#if !defined()` OR `#ifndef` all need to be paired with `#endif`

- `#define _STD_LIB_H:` creates the macro `_STD_LIB_H.`

- Compilation: See `1_intro_to_terminal_edit_file.pdf`

C++ Input and type

- Input must be compatible with type.
  - That is, what is read in must be convertible to the type being stored.

- Incompatible data will be rejected, the read fails, and the data is left on the stream!!!

- The `>>` extraction operator parses (by default) based on 'white space'. So, reading text with spaces in it requires a different function, .e.g. `getline(...), get()`
  - http://www.cplusplus.com/reference/string/string/getline/

Example

```
int main()
{
        cout << "Enter first and age\n";
        string first_name;
        int age;
        cin >> first_name;
        cin >> age;
        cout << "Hello, "<< first_name << "(age "<<age<< ")\n";
}
```

What happens if you put in `34`, `Dani`?

Array (data aggregate)

- https://www.geeksforgeeks.org/arrays-in-c-cpp/
- https://www.tutorialspoint.com/cplusplus/cpp_arrays.htm
- C/C++ provides a data structure, the array, which stores a fixed-size sequential collection of elements of the same type. All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.
- Syntax of declaration: `data_type arrayName[arraySize];`

`data_type`: Type of data to be stored in the array. Here `data_type` is valid C/C++ data type
`arrayName`: Name of the array
`arraySize`: Sizes of the array, and must be an integer constant greater than zero

```
int main(){
 double balance[10]; //Declaring Array
 double balance2[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
  // declaration and initialization

  balance2[0] = 100.0;//all arrays have 0 as the index
  //of their first element which is also called base index.
  balance2[4] = -2.0;// this changes the 5th element of array
}
```

Another example

- https://www.tutorialspoint.com/cplusplus/cpp_arrays.htm

- One common “bad” practice
  - Variable length arrays (VLA) are not standardized in C++. They are up to compiler vendors to implement. Can potentially cause memory related issues for large size array (run out of stack).

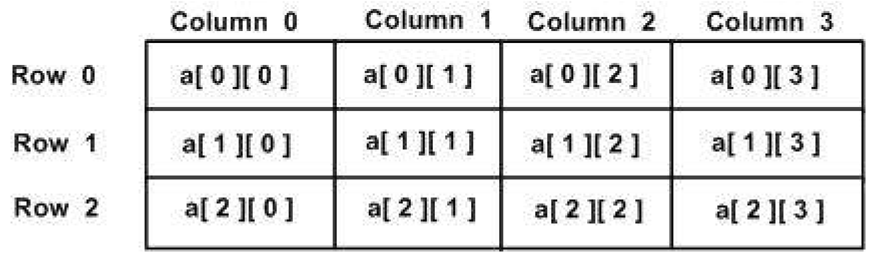
```
~zxu2/Public/variable_array_size.cpp
```

Multi-dimensional array

- Syntax: `data_type array_name[size1][size2]…[sizeN];`

  `data_type`: Type of data to be stored in the array. Here `data_type` is valid C/C++ data type
  `array_name`: Name of the array
  `size1, size2,... ,sizeN`: Sizes of the dimensions

- Example.



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

```
int main(){
int b[2][2],
    a[3][4] = {
   {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
   {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
   {8, 9, 10, 11}}; /*  initializers for row indexed by 2 */
   for ( int i = 0; i < 3; i++ )
      for ( int j = 0; j < 4; j++ ) {

         cout << "a[" << i << "][" << j << "]: ";
         cout << a[i][j]<< endl;
      }
}
```

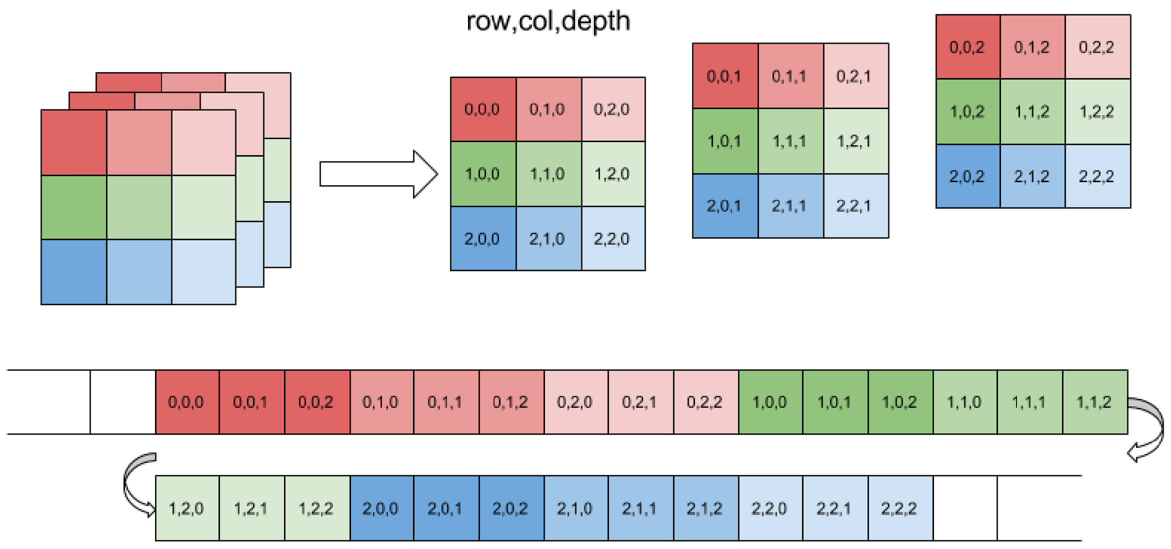
- https://www.tutorialspoint.com/cplusplus/cpp_multi_dimensional_arrays.htm

Multi-D arrays memory storage

1. C/C++ stores a multi-dimensional (rectangular) array in a contiguous memory area.
2. C/C++ uses *row-major layout*. When 2D array (i.e. matrix) is stored, the first row is stored first, then the second row, the third row then follows and so on.

a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[2][0]	a[2][1]	a[2][

3. 3D array is similar



Strings

- Used to store text.

- C-style strings:
  - Array of characters
  - `\0` terminated.
  - Functions provided in `<cstring>`
  - `char cstr[256]; // declare`

- C++ string:
  - Described in `<string>`

```
string firstName = "John"; string lastName = "Smith";
string name = firstName+ " "+ lastName;
```

Operations and operators

- https://www.tutorialspoint.com/cplusplus/cpp_operators.htm
- Type implies the operations that can be done to a variable(s). e.g. `+' means different things to different types

-
  ```
  int a = 0;
  int b = a+2;
  int c = a-2;
  ```

-
  ```
  string s1 = "dani";
  string s2 = s1 + " brake";
  string s3 = s1 - "i"; // error, undefined
                        //operator for string
  ```

operators

- An operator is a function with a symbol name. (Some) operators can be overloaded (redefine to give new functionality.)

- Here are some examples
  - +
  - -
  - *
  - /
  - =
  - %
  - %=
  - +=
  - ++
  - --

  - <
  - <=
  - &
  - &&
  - |
  - ||
  - !=
  - !
  - /=
  - <<
  - >>

  - The operators have a well-defined order of operation. This can easily be found online or in the book.
  - https://www.geeksforgeeks.org/operators-c-c/
  - https://www.tutorialspoint.com/cplusplus/cpp_operators.htm

assignment

- The function `=` is called the 'assignment operator'
- It replaces the value of the left hand side object with whatever is computed on the right hand side.
  ```
  name = first_name + ' ' + last_name;
  ```
- It can be combined with creation of a variable.
  ```
  string name = first_name + ' ' + last_name;
  ```

when a variable appears on both sides of =

- Let's dissect the line

```
a = a+7;
```

- This does not mean that `a` is the same as `a+7`. Nope.
- Instead, it means that the value `a+7` should be calculated.
- Then, this new value replaces the old value of `a`.
- That is, this line increases the value of `a` by 7.

Composite assignment

- There are several operators which modify the value of a variable.

- E.g. `++` (increment), `--` (decrement)
  - Increase or decrease by 1.

```
int a=0;
a++; // now a has value 1.
```

  - Frequently used by `int`, pointer and iterator variables.

- E.g. `+=`, `-=`, `*=`, `/=`

```
double x = 1.5;
x *= 4; // multiply x by 4, and store back into x
```

Additional readings

- Tim Love, ANSI C for Programmers on UNIX Systems (online)
- https://www.tutorialspoint.com/cplusplus/index.htm
  (C++ overview ---- C++ Strings)

- Bjarne Stroustrup, The C++ Programming Language
  (Part I, Part II)

Sec.2 STATEMENTS AND FLOW CONTROL

Agenda

- Statements (or constructs)
  - Programming concepts are implemented by language provided constructs
  - http://www.cplusplus.com/doc/tutorial/control/

Statements

- A (expression) statement is a thing which comes before the `;`

- The simplest statement: the do-nothing statement:

```
;
NULL; //recommended way
```

- Other really simple statements, which do nothing:
  - `1+2;`
  - `a*b;`

- Statements are executed sequentially, except when an expression statement, a selection statement, an iteration statement, or a jump statement *specifically modifies* that sequence.

- Here we briefly cover: selection, iteration, jump statements.

Code blocks

- Code can be put into a code block (compound statements). Blocks are denoted by `{ }` in C++
- *any new variables made in a block disappear at the end (with exception:* `static`*)*
- changes made to variables created before the block still have taken place.
- use them in `` `if` ``, `` `else` ``, `` `while` ``, `` `for` ``, `` `do-while` ``, `` `switch` ``, function bodies, and many other places
- I also call them '[scopes]'

if

- We can select what happens, based on conditionals.
- The keyword is `` `if` ``. Here are some rules:

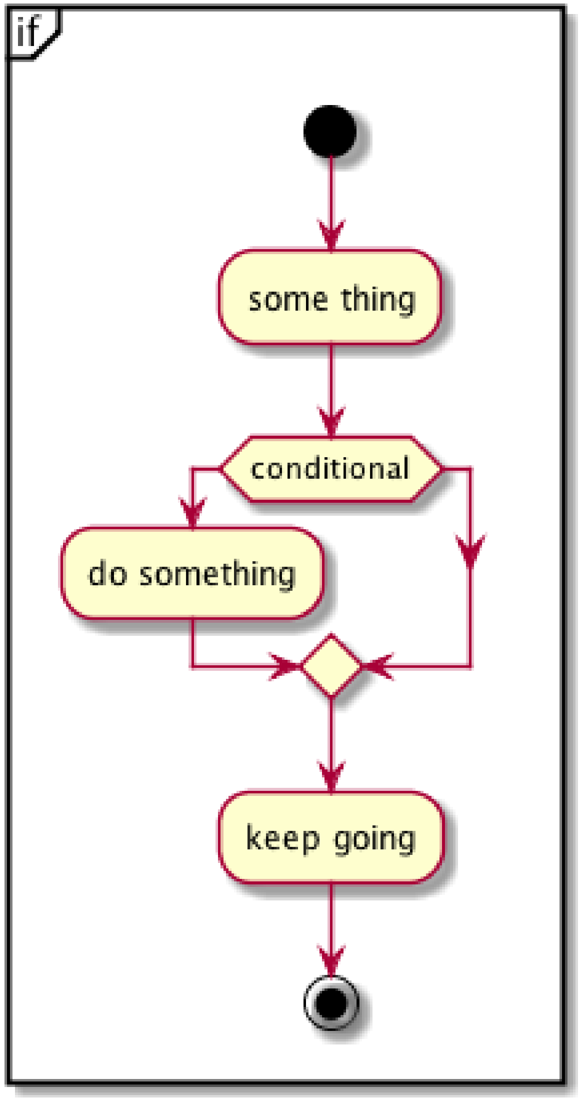
```
if(boolean_expression) {
    // statement(s) will execute if the boolean
    // expression is true
}
```

- `` `if` `` has to have (...) behind it. The `()` are not optional.
- `` `if` `` can have one or more things after it.
  - Just one thing? no need for curly braces.
  - more than one thing? use curly braces
- the thing after `` `if` `` happens if the thing in `(...)` is true.
  - whatever is in the `()` gets converted to true or false.
- want something to happen in false case?
  - use `` `else` ``. same rules as `` `if` `` for number of things.

  `1;   {>1;}` //run either one statement or a block of them.
- https://www.tutorialspoint.com/cplusplus/cpp_decision_making.htm

if

- Optional execution of a block of code
- Only do contents of the block when conditional is true.



A surprise for you!

- Surprising empty statement:

```
if (x==5); // if you really mean this, implement as:
{y=3;}   // empty statement behind the if.  y=3;
         // happens all the time!
```

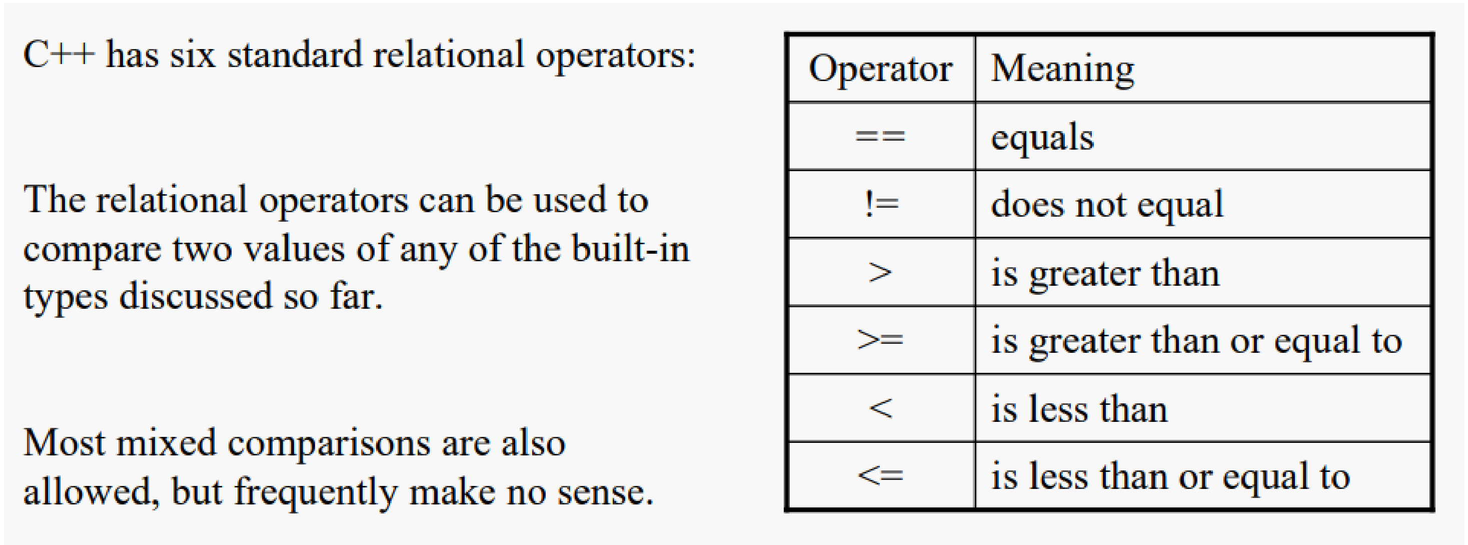
watch out for those stray semicolons!!!

```
// if you really mean this, implement as:
// See Sec. 7.7 in Appendix_1_Writing_clean_code.pdf
```

C++ Boolean expression

- `bool` type variables can have either of two values: `true, false`
- To ask a question, a C++ program uses a boolean expression which is evaluated to either true or false at run-time.
- Example. Assert “The student’s age is above or equal to 21?”:

```
const int LEGAL_AGE = 21;
bool is_Legal_Age;
int    stuAge;
cin>> stuAge;
is_Legal_Age = (stuAge >= LEGAL_AGE);
```



C++ has six standard relational operators:

The relational operators can be used to compare two values of any of the built-in types discussed so far.

Most mixed comparisons are also allowed, but frequently make no sense.

Operator	Meaning
==	equals
!=	does not equal
>	is greater than
>=	is greater than or equal to
<	is less than
<=	is less than or equal to

C boolean

- C does not have boolean data types, and normally uses integers for boolean testing.

- This was done using `#define`. Somewhere in the code, we see:

```
#define true 1
#define false 0
```

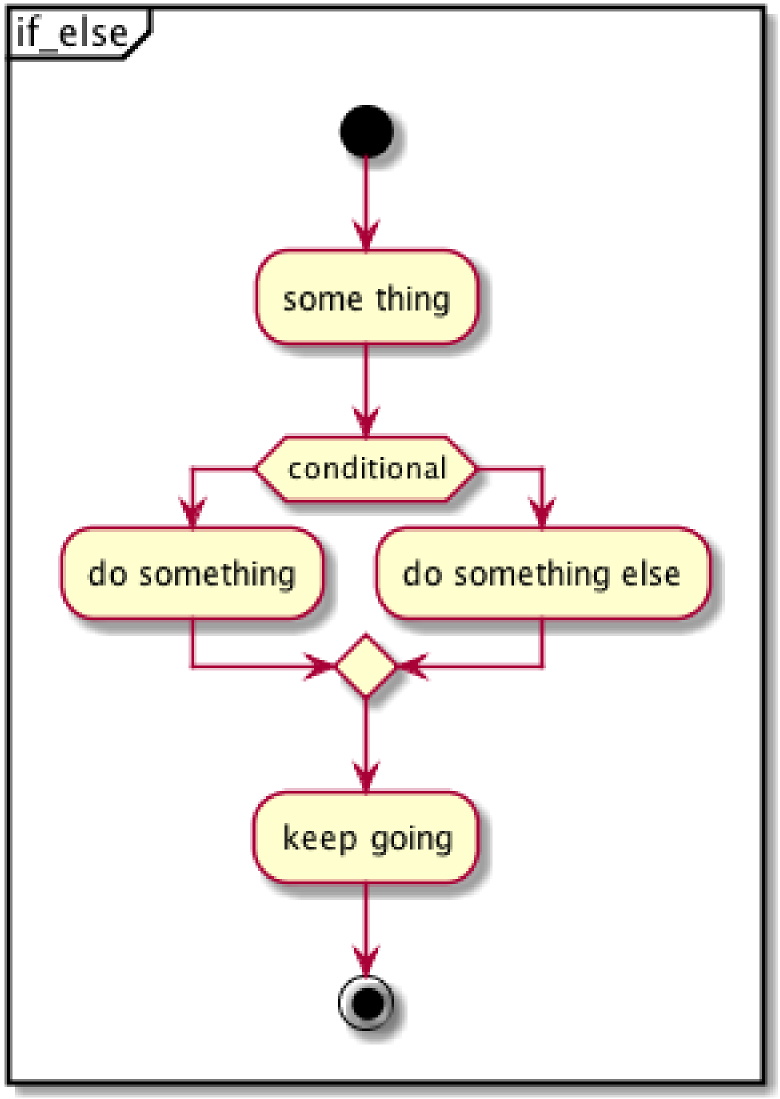
- Or we can use `const int` instead of `#define`. Somewhere in the code, we see:

```
const int true = 1;
const int false = 0;
```

if else

- One of two things guaranteed to happen
- Either conditional true, or conditional false. One of those two things has to be true.
- Provides two alternative paths of execution

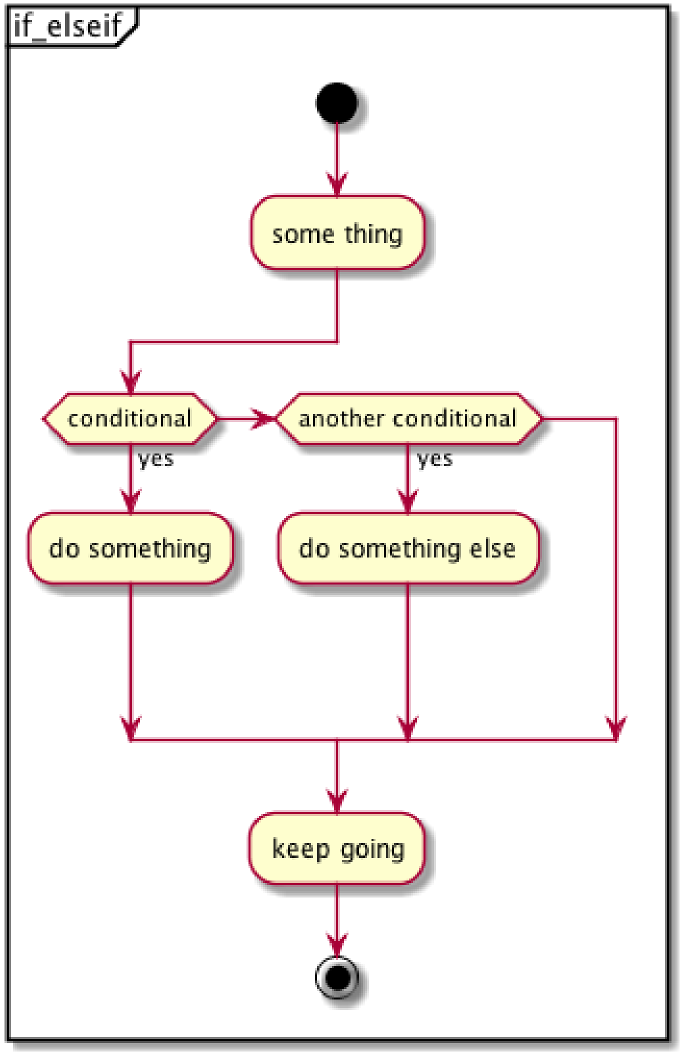
```
if(boolean_expression) {
   // statement(s) will
// execute if the Boolean
// expression is true
} else {
  // statement(s) will
// execute if the boolean
// expression is false
}
```



if else-if

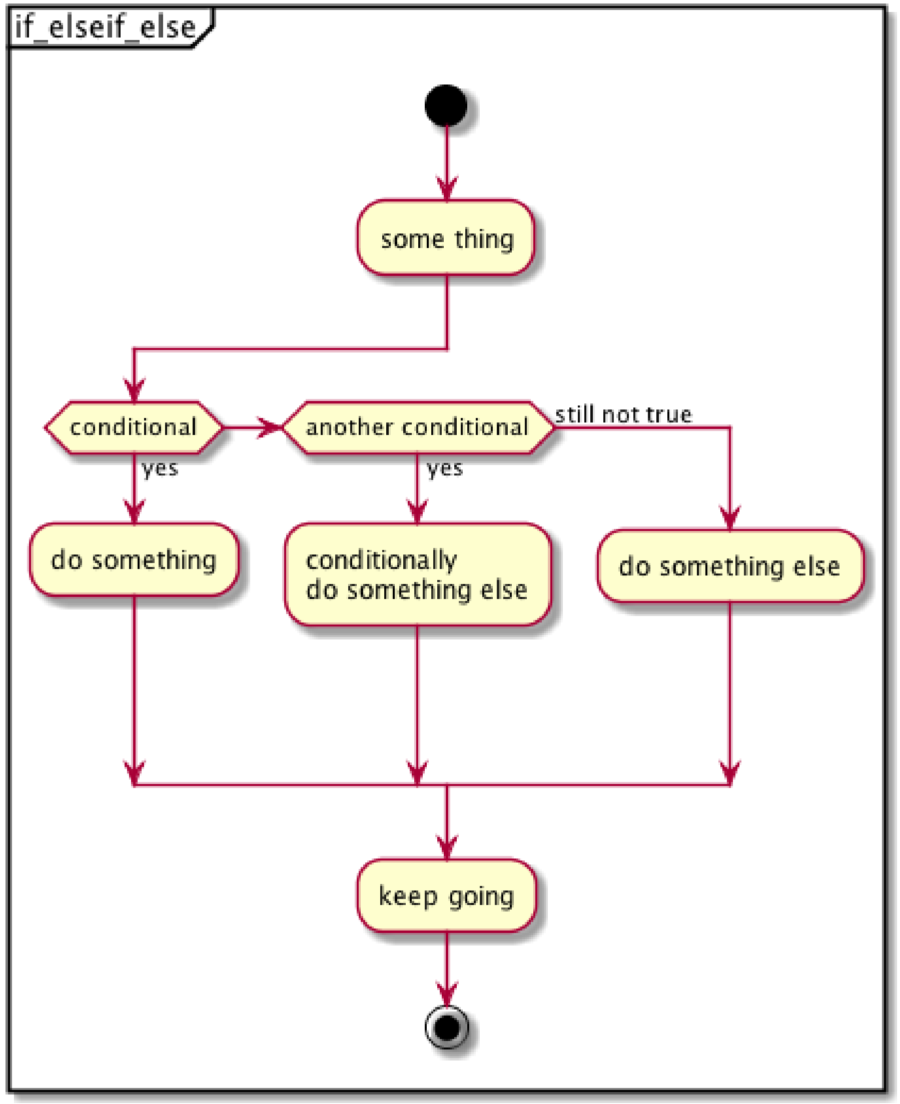
- Nothing guaranteed to happen
- If neither conditional true, then nothing happens
- This is because there is no `else` branch

```
if(boolean_expression) {
    // statement(s) will
// execute if the boolean
//expression is true
} else if(boolean_exp2) {
   // statement(s) will
//execute if the
//boolean_exp2 is true
}
```



if else-if else

- One of three things has to happen



Learn from Example

- https://www.tutorialspoint.com/cplusplus/cpp_if_else_statement.htm

Iteration

- To do things multiple times, we iterate

- Iteration comes with several keywords in C++
  - `do while`
  - `while`
  - `for`

- There are other ways, too (C++11)
  - `std::for_each(...)    // STL alg.`

for

- `` `for` `` loop consists of about 4 pieces:
    1. loop counter
    2. terminating condition
    3. incrementing expression
    4. statements to be repeated

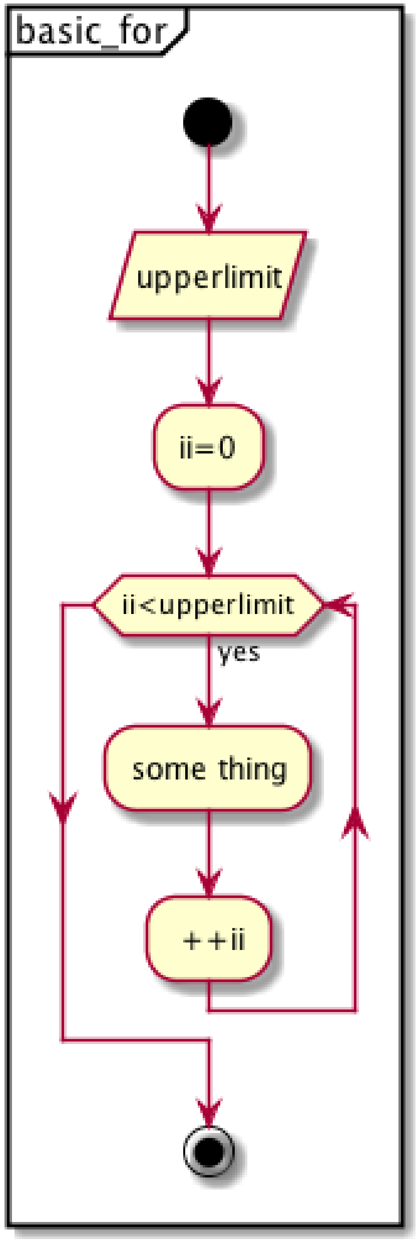
- similar syntax to `` `while` ``.
    - round braces () come after `` `for` ``. Not optional. Also, need two semicolons. `for (`*`counter; condition; increment`*`)`

      `1;    {>1;}` //run either one statement or a block of them.

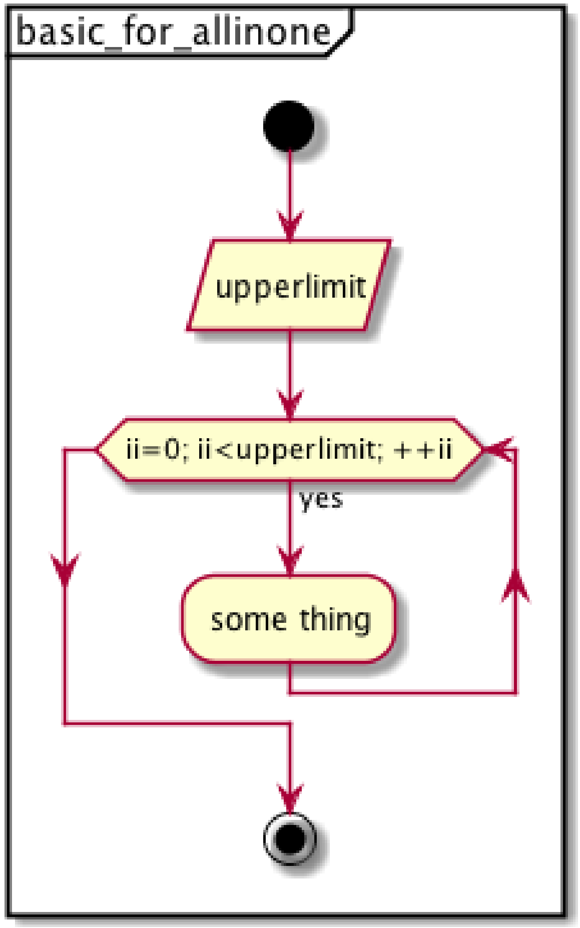
https://www.tutorialspoint.com/cplusplus/cpp_loop_types.htm

Basic for loop

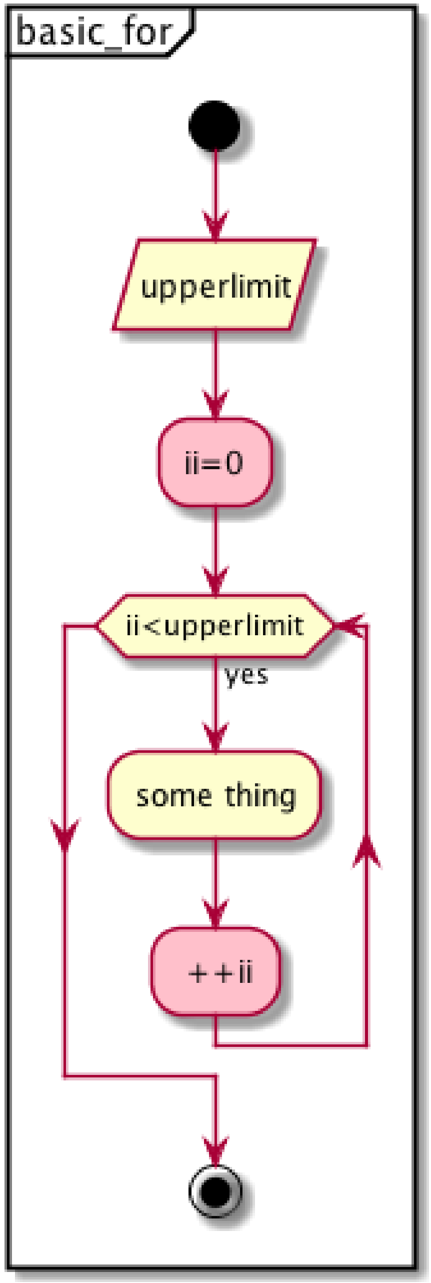
- Repeat a given number of times
- Requires
  - a thing to keep track of number of times – this is the ”loop counter”. I call it `ii` usually
  - A number of times to repeat
  - A call to increment loop counter this is `++ii` often



A contraction



Nice and concise



a little
more verbose

example - `for`

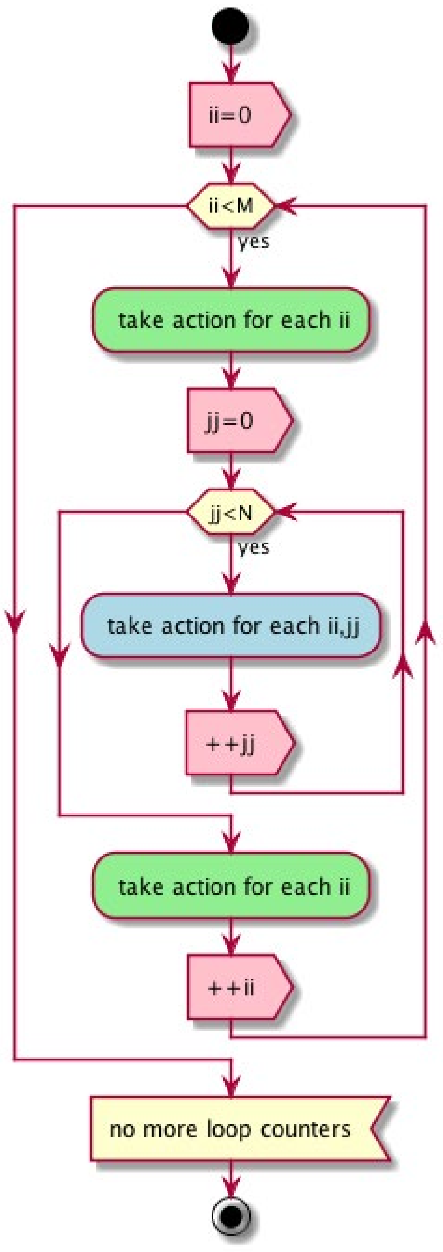
```
int main() {
      constexpr double pi = acos(-1);
      for (int ii=0; ii<100; ++ii)
      {
            cout << "tan(" << pi/ii << ") = ";
            cout << tan(pi/ii) << '\n';
      }
}
```

- `constexpr` is a feature added in C++ 11. The main idea is performance improvement of programs by doing computations at compile time rather than run time.
- https://www.geeksforgeeks.org/understanding-constexper-specifier-in-c/

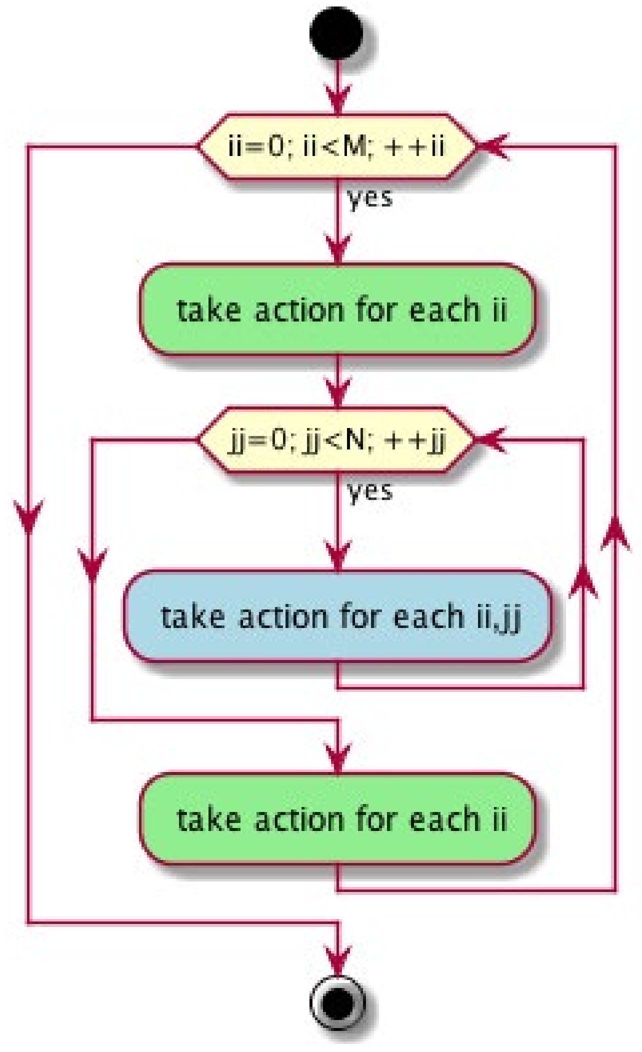
double for loop

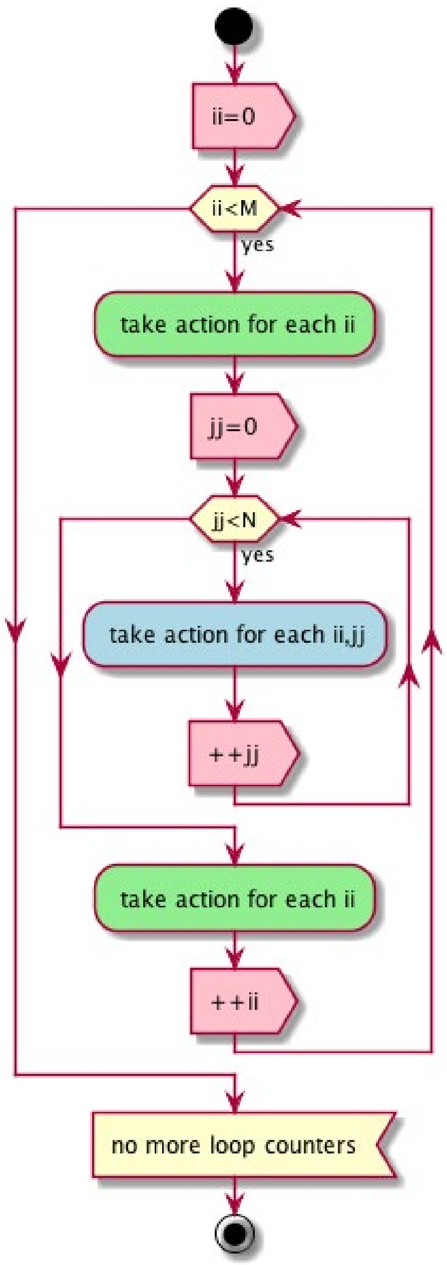
- inner loop executes for each iteration of the outer loop
- this depiction explicit on when things like `ii=0` and `++ii` happen

- Example:
  - Matrix matrix multiplication
  - Matrix vector multiplication
  - https://www.geeksforgeeks.org/c-program-multiply-two-matrices/



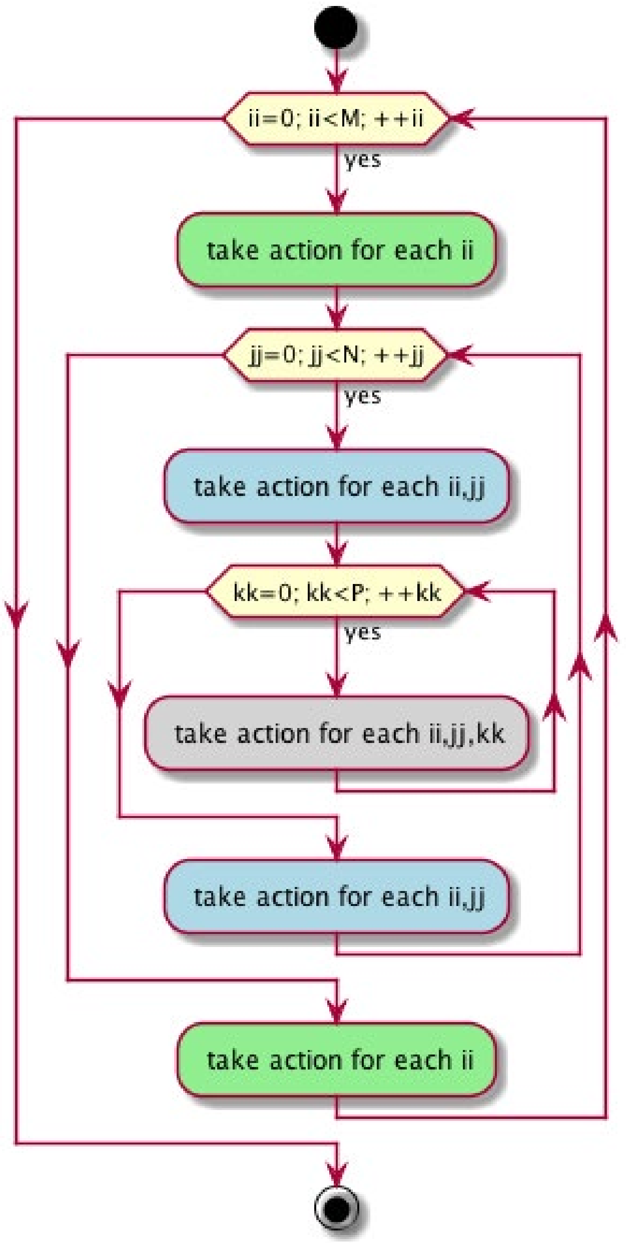
double for – verbose vs compact





a triple for loop

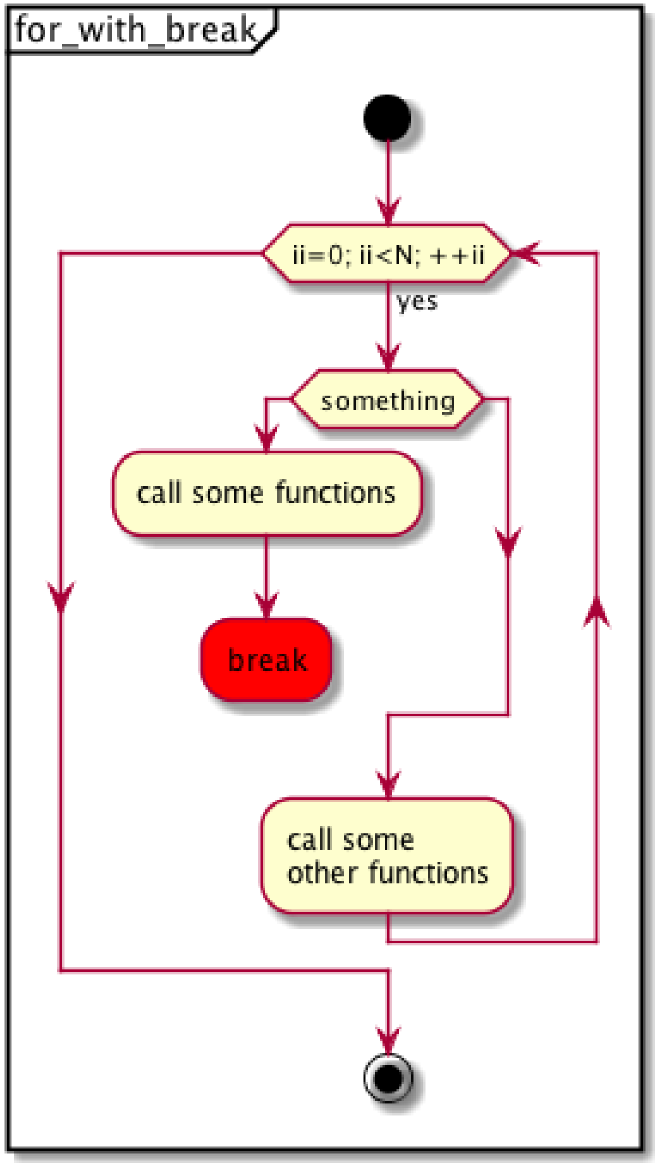
- if you're in triple for loop land, consider factoring out inner loops into their own function.
- avoid nesting over two levels deep



combining

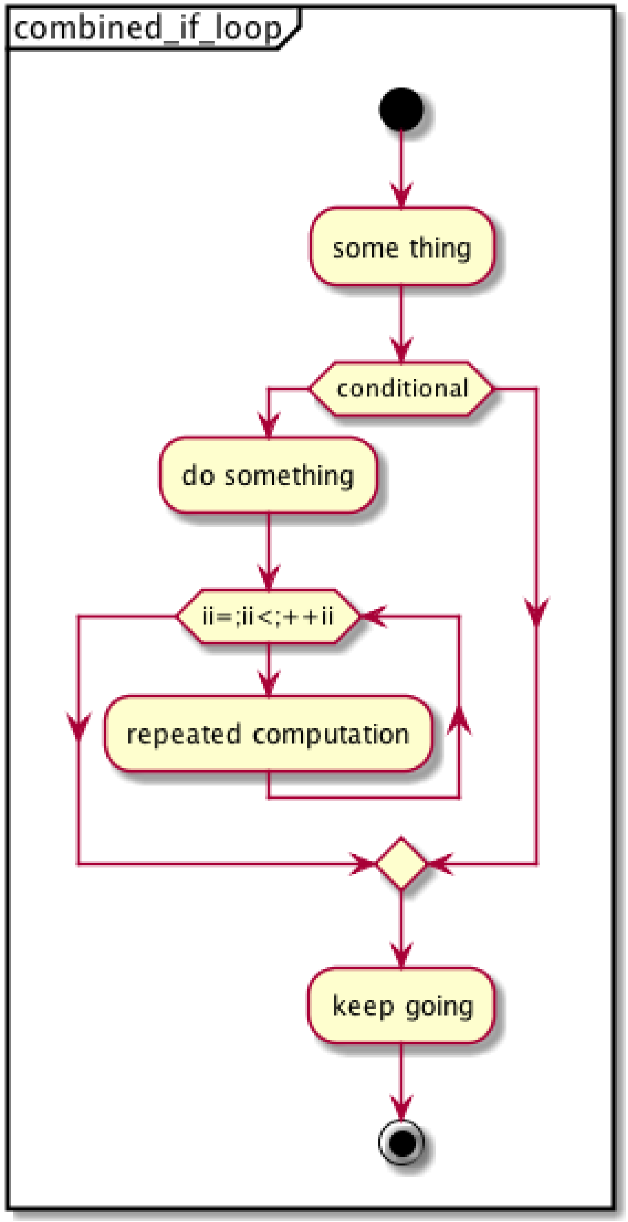
- using an `if` inside a loop, you can optionally terminate early.
- this uses the c++ keyword `break`

https://www.tutorialspoint.com/cplusplus/cpp_nested_loops.htm



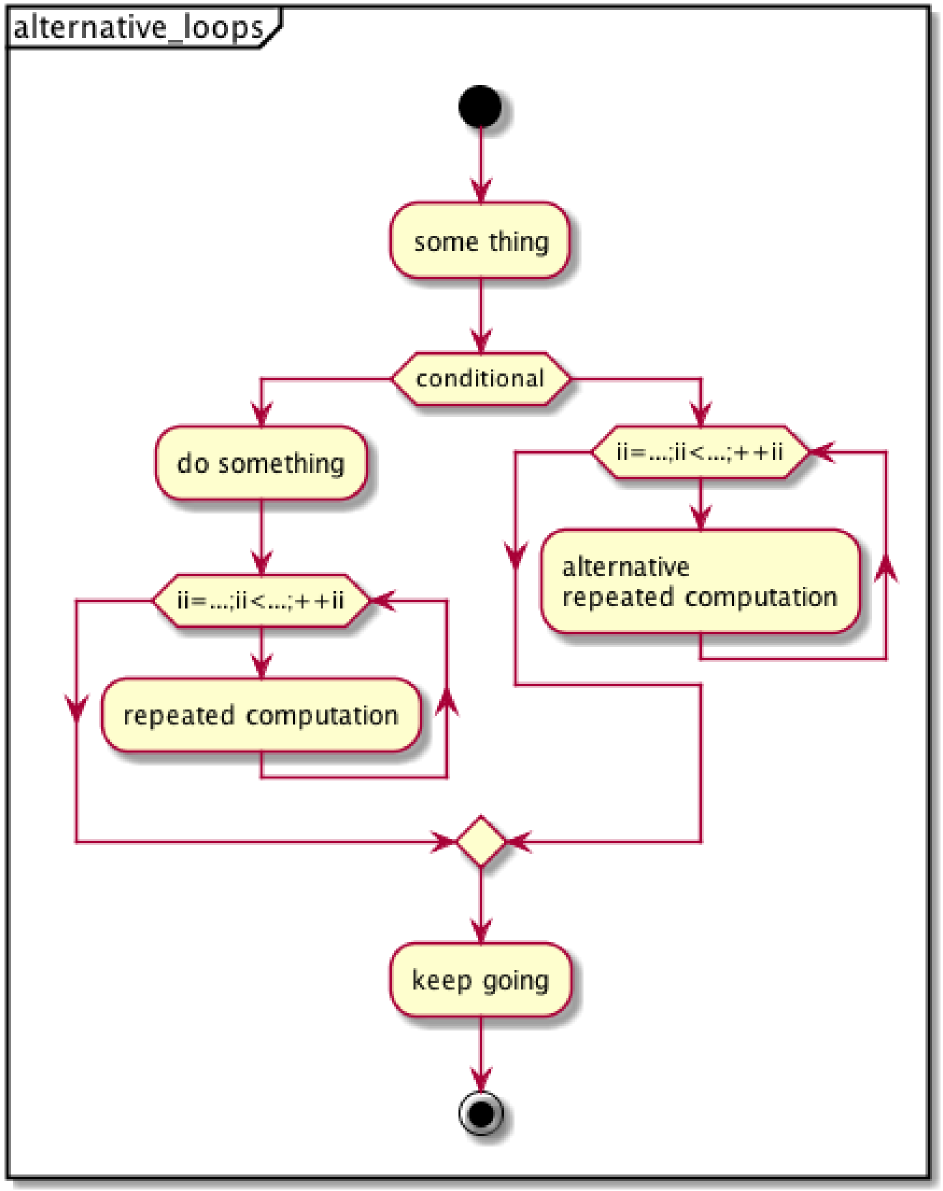
looping inside an `if`

- you can do arbitrary things inside an `if` or `for`.
- like, put a loop in a `for`
- or, put an `if` in a loop
- if things get complicated, rewrite as functions, and call those functions.



two alternative loops

- in this diagram, one of two loops will run, depending on a conditional.
- it's ok, and good, to use the same letter as counter for both loops.
- scope keeps the use of the variables separate



while

- \`while\` lets you do things until some expression evaluates to false.
- just like `if, else`, the `while` loop can run either a single statement, or a block of them.

```
        1;   {>1;}
```

```
while (expression here)
// stops when this expr is false.
{
     multiple statements here;
}
```

// code down here doesn't repeat in the above while loop

example - while

```
int main() {
     int i=0;
     while (i<100) {
          cout << i << '\t' << square(i) << '\n';
          ++i; // increment i
     } // ends the while
}
```

components:

- something to become false,
- something to do.

switch

- There's another way to branch. Have many things to do, many 'cases', so to speak? Use a `switch`!

- ‘switch’ lets us decide what to do based on a finite number of known constant possibilities (at compile time).

- Cases described with a colon :
- Use `break` to terminate cases.
- case labels must be constant (known at compile time)

Example

```
…
switch (exp){
      case 'a':
            code here;
            break; // stop doing case 'a'
      case 'b':
            code here
            break;  // stop doing case 'b'
      default:
            code here
            break;
}
```

The `exp` used in a **`switch`** statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.

https://www.tutorialspoint.com/cplusplus/cpp_switch_statement.htm

https://www.geeksforgeeks.org/enumerated-types-or-enums-in-c/

notes on switch

- `switch` nice for when have many `if`s. Get more readable code. However,
- Cases must be known at compile time!!
- Cases must be convertible to `int`.
  Sorry, no string switches like in Matlab.

- Omitting a `break` causes a case fall-through.
  Fall through happens until you reach a break or the end of the switch.

- example

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

int main () {
   // local variable declaration:
   char grade = 'D';

   switch(grade) {
      case 'A' :
         cout << "Excellent!" << endl;
         break;
      case 'B' :
      case 'C' :
         cout << "Well done" << endl;
         break;
      case 'D' :
         cout << "You passed" << endl;
         break;
      case 'F' :
         cout << "Better try again" << endl;
         break;
      default :
         cout << "Invalid grade" << endl;
   }
   cout << "Your grade is " << grade << endl;

   return 0;
} // see also https://www.geeksforgeeks.org/switch-statement-cc/
```

break

- Causes the enclosing for while or do-while loop or switch statement to terminate.

```
#include <iostream>

int main()
{
    int i = 2;
    switch (i) {
        case 1: std::cout << "1";
        case 2: std::cout << "2";   //execution starts at this case label
        case 3: std::cout << "3";
        case 4:
        case 5: std::cout << "45";
                break;      //execution of subsequent statements is terminated
        case 6: std::cout << "6";
    }

    std::cout << '\n';

    for (int j = 0; j < 2; j++) {
        for (int k = 0; k < 5; k++) {    //only this loop is affected by break
            if (k == 2) break;
            std::cout << j << k << " ";
        }
    }
}
```

continue

- Causes the remaining portion of the enclosing for, while or do-while loop body to be skipped.

```
#include <iostream>
int main()
{
    for (int i = 0; i < 10; i++) {
        if (i != 5) continue;
        std::cout << i << " ";//this statement is skipped each time i!=5
    }

    std::cout << '\n';

    for (int j = 0; j < 2; j++) {
      for (int k = 0; k < 5; k++) { //only this loop is affected by continue
        if (k == 3) continue;
        std::cout << j << k << " "; //this statement is skipped each time k==3
        }
    }
}
```

return

- Terminates the current function and returns the specified value (if any) to its caller.

```
int fb(int i)
{
    if (i > 4)
         return 4;
    std::cout << i << '\n';
    return 2;
}
int main()
{
    int I = fb(5); // returns 4
    I = fb(I); // prints its argument, then returns 2
}
```

Quizzes

- What will the code below print out?

```
#include <iostream>
int main(){
    std::cout<< 25-50;
}
```

- What will `i` and `j` equal?

```
int i = 5;
int j = i++;
```

- Find any errors in the following C++ program:

```
#include<iostream>
using namespace std;
void main(){
/* This is the main function*/
cout<<"Hello"
return 0;
}
```

- Find errors in the following C++ program:

```
#include<iostream>
using namespace std;
void main(){ /* This is the main function
  cout<<'C++ Programming questions and answers';
}
```

- Which of the following is not a logical operator in C++ language?

```
     a. &   b. &&   c. ||   d. !
```

- Evaluate the following expressions to true or false

```
    a. 3!=4-1   b. 5>=(1+6)-4   c.5+1==6 || 8-1>4
```

- If originally `x=0,y=0,` and `z=1`, what is the value of `x,y,` and `z` after executing the following code?

```
if(x) {x--;y=x+2;}
else x++;
```

- If originally x=1,y=0, and z=1, what is the value of x, y, and z after executing the following code?

```
if(x>y && x>z) {
    y=x;z=x+1;
}
else if(x+y>=z) {
    x++;z=x+1;
}
else
    y=z+x;
```

- Rewrite the following C++ code using switch statement:

```
if(ch=='a' || ch=='A') countA++;
else if(ch=='e' || ch=='E') countE++;
else if(ch=='i' || ch=='I') countI++;
else if(ch=='o' || ch=='O') countO++;
else if(ch=='u' || ch=='U') countU++;
else cout<<"No vowel letter";
```

- Change the following C++ code from using a while loop to for loop:

```
x=1;
while(x<10){
  cout<<x<<"\t";
  ++x;
}
```

- Change the following C++ code from a while loop to a for loop:

```
int x;
cin>>x;
while(x!=10){
  cout<<x<<"\t";
  cin>>x;
}
```

- What would be printed from the following C++ code segment?

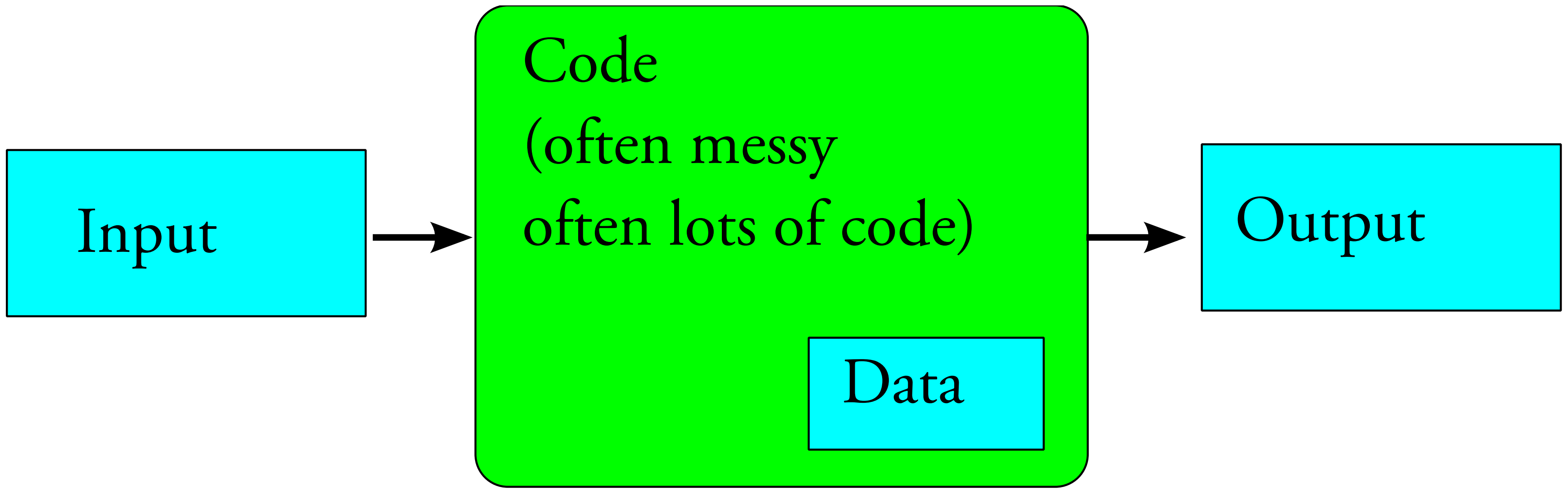
```
for(x=1;x<=5;x++){
    for(y=1;y<=x;y++)
        cout<<x<<"\t";
    cout<<"\n";
}
```

Sec. 3 COMPUTATION

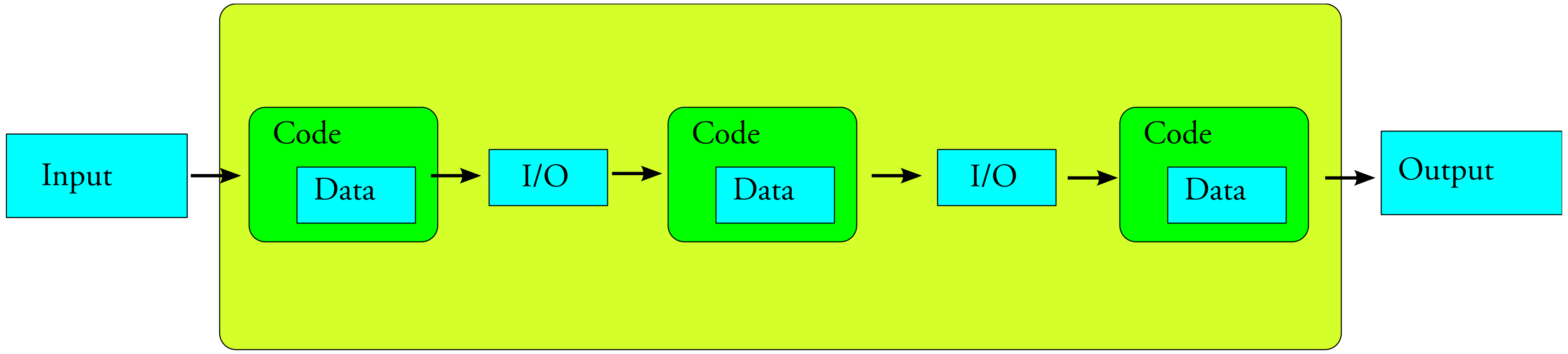
Agenda

- C++ model of computation
- Programming ideas

A super simple model



A more complicated model



Objectives in programming

Our main objectives are to program

1. correctly
   incorrect code is garbage
2. simply
   complicated code is hard to read and maintain
3. efficiently
   inefficient code wastes time

Practices to achieve goals

- Abstraction
  - Hide details behind an interface.
  - Functions and classes are our main tools of abstraction.
- Divide and Conquer
  - Break a large problem into smaller problems.
  - Keep breaking down until you get to small units of work
  - Solve these units of work.
  - Test and debug
  - Integrate
- Do dry run many many times
- Documentation
- …

A closer look at Gaussian_eli.cpp

- `~zxu2/Public/homework_01`
- The code implement Gaussian elimination with backward substitution for solving system of linear equations. (see also `Sakai/Resources/Project 01/Gaussian-elimination-alg.pdf`)

- Problems associated with the code:
  1. Not easy to adjust problem sizes.
  2. Bad abstraction (essentially a single `main()` function)
  3. I/O?
  4. …

Sec. 4 DYNAMIC MEMORY ALLOCATION, POINTER, REFERENCE

Agenda

- Pointer
  - A mechanism for referring to memory
- Dynamic memory allocation
  - C: `malloc(), free();` C++: `new, delete`
- Reference

Pointer

- A variable is a memory location which can be accessed by the identifier (the name of the variable).
- In C/C++, variables and objects “have identity”. That is, they reside at a specific address in memory, and a variable/an object can be addressed if its memory address and its type are known.
- The language constructs for holding and using addresses are pointers and (references, a C++ feature).

- `int` *k*; /* the compiler sets aside 4 bytes of memory (on a PC) to hold the value of the integer. It also sets up a symbol table. In that table it adds the symbol *k* and the relative address in memory where those 4 bytes were set aside. */

- *k* `= 8`; /*at run time when this statement is executed, the value 8 will be placed in that memory location reserved for the storage of the value of k. */

- With *k*, there are two associated values. One is the value of the integer, 8, stored. The other is the “value” or address of the memory location.

- The variable for holding an address is a pointer variable.
- Syntax for defining a pointer: `T* pointer_name;`
    - Here T is a C/C++ type.

`int* ptr;` /*we also give pointer a type which refers to the type of data stored at the address that we will store in the pointer. "*" means pointer to */

```
int k ;
k = 8;
```

`ptr = &k;` /* `&` address-of operator retrieves the address of *k* */

//See code:
`Public/ACMS40212/C_basics/pointer_basics.cpp`

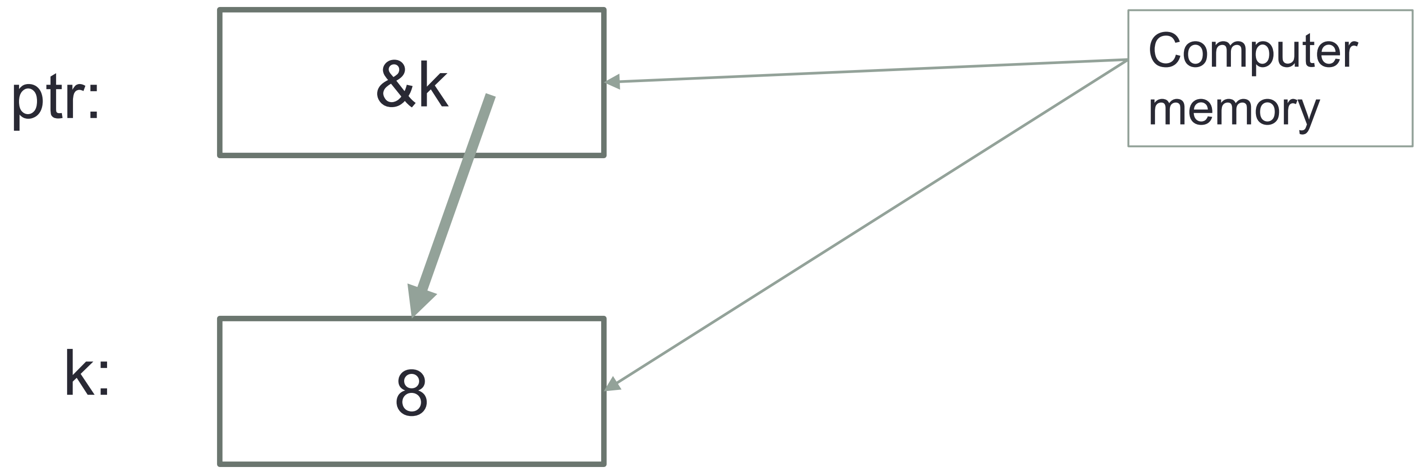
// Types of pointer and variables it points to must match

- A fundamental operation on a pointer is dereferencing, that is, referring to the object pointed to by the pointer.
- The dereferencing operator is (prefix) `*`.

```
   int* ptr = nullptr;
   int k ;
   k = 8;

   ptr = &k;
    *ptr = 7;
```

/* dereferencing operator "`*`" copies 7 to the memory address pointed to by `ptr` */

Graphically,



More examples

- `int*    pi;` // pointer to `int`
- `char** ppc;` // pointer to "pointer to char"
- `int*    ap[15];` // array of 15 pointers to `int`s
- `int   (*fp)(char*);` // pointer to function taking a `char*`
  // argument; returns an `int`.
- `int*   foo(char*);` // function taking a `char*` argument;
  // returns a pointer to `int`.
- `void* pv;` // pointer to an object of unknown type.
  // usually used in very low-level code.
- `nullptr`: //Literal representing the null pointer, that is, a pointer
  //that does not point to an object.

Pointer into array

- An array represents a sequence of objects in memory contiguously.
  - `T[size]`. The elements are indexed from 0 to size-1.
  - `double a[3];` // an array of three doubles: a[0], a[1], a[2]

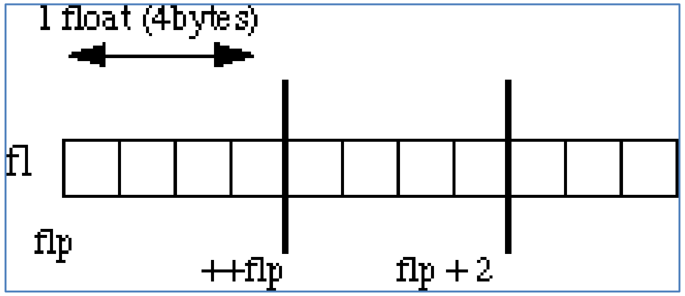
1. The name of an array is a constant pointer to its initial elements.
2. Four arithmetic operations can be performed on pointers: ++, --, +, -.
   - ++, +: Each time a pointer is incremented by 1, it points to the next element.
   - The array name cannot be incremented because it is a constant pointer.

```
float a[100], *pfl;
pfl = &(a[0]); /* or pfl = a;*/ // Point pfl to the first element, a[0]
pfl = &(a[3]);/* or pfl = a + 3; */ // Point pfl to the 4th element
```

Navigating array

```
float  a[100],  *flp;
flp = a;
/* now increment flp to point to successive elements */
for(int i =0; i < 100; i++)
{
   cout<< "*flp is = "<< *flp <<endl;
   flp++;  /*or flp += 1; */ // flp is incremented by the length of an float
                             // and points to the next float, a[1], a[2] etc.
}
```

See code: ACMS40212/C_basics/pointer_array.cpp

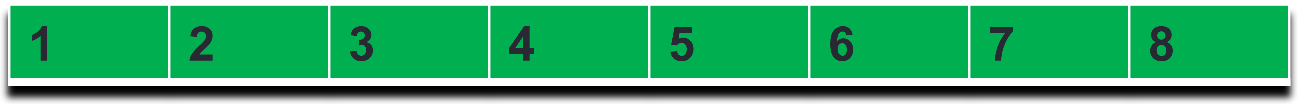


Never go beyond range!

```
float  a[100],  *flp;
flp = a;
float* flp2 = flp -2; // result is undefined.
float* flp3 = &a[99]+1; // result is undefined.
```

- Multidimensional arrays and pointers

1. Array elements are stored row by row. Hence a 2D array (i.e. `int myMatrix[2][4] ={{1,2,3,4}, {5,6,7,8}};`) is really a 1D array, each of whose element is itself an array.
   - `int myMatrix[2][4] ;` // 2 rows of 4 ints each.
   - Graphically,



2. `myMatrix`: pointer to the 0th row of the 2D array (or the 0th 1D array element);

   `myMatrix[0]`: pointer to (address of) the 0th row of the 2D array;

   `myMatrix[1]`: pointer to the 1st row of the 2D array.

   `myMatrix+1`: pointer to the 1st row element of the 1D array.

3. `myMatrix[i][j]` is same as: `*(myMatrix[i] + j)`,

   `*(&myMatrix[0][0]+4*i+j)`, `(*(myMatrix + i))[j]`, and … // ACMS40212/C_basics/pointer_2dmatrix.cpp

4. https://www.geeksforgeeks.org/pointer-array-array-pointer/

Pointer and text strings

- Text strings in C have been implemented as arrays of characters, with the last byte in the string being a zero, or the null character '\0'.

```
static const char *myFormat = "Total Amount Due: %d";
```

//The variable myFormat can be viewed as an array of 21 characters.

- An initialized array of strings would typically be done as follows:

```
static const char *myColors[] = {
   "Red", "Orange", "Yellow", "Green", "Blue",
"Violet"};
```

Dynamic memory allocation in C++, C

In the programs seen so far, all memory needs were determined before program execution by defining the variables needed.

- Recall Gaussian elimination code.

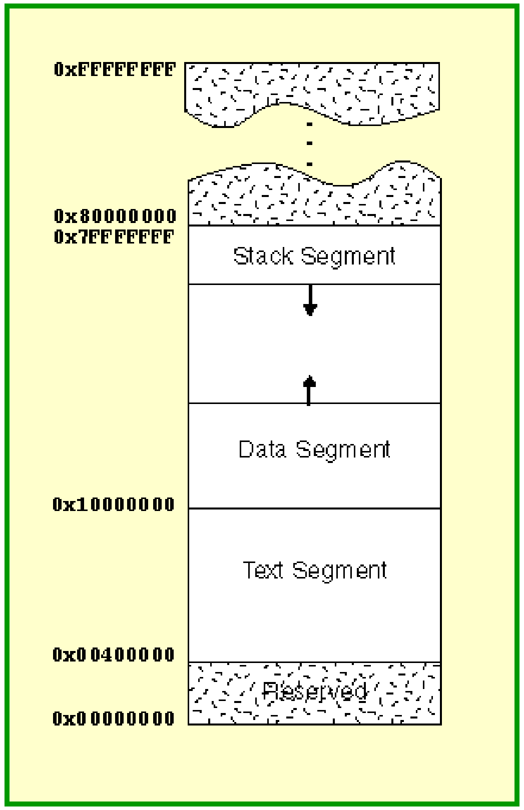
There are occasions:

- The size of the problem often can not be determined at “compile time”.

When this is the case,

- Dynamic memory allocation is needed to allocate memory at “run time”.
- Dynamically allocated memory must be referred to by pointers.

Stack vs Heap



When a program is loaded into memory:

- Machine code is loaded into **text** segment
- **Data** segment: global, static and constants data are stored in the data segment
- **Stack** segment allocates memory for automatic variables within functions, and is used for passing arguments to the function etc.
- **Heap** segment is for dynamic memory allocation
- The size of the text and data segments are known as soon as compilation is completed. The stack and heap segments grow and shrink during program execution.

C++ operators: `new, new[], delete, delete[]`

- Objects allocated by new are said to be “on the free store”, or “on the heap memory”.
- The operator delete is used to destroy them.
- Syntax to use `new`: `pointer-variable = new type;`
- Syntax to use `delete`: `delete pointer-variable;`
- `#include <new>`

```
// Pointer initialized with null pointer
// Then request memory for the variable
double *ptr = nullptr;
ptr = new double;
*ptr = -2.5;  // -2.5 is stored.
delete ptr; // release allocated memory
/*Once dynamic memory is freed, do not do things like: *ptr = 5.0;*/
```

- graphically

Name	Type	Contents	Address
ptr	double pointer	0x3D3B38	0x22FB66

Memory heap (free storage we can use)	
…	
0x3D3B38	-2.5
0x3D3B39	

- Syntax to use `new[]`: `pointer-variable = new type[size];`
  - This allocates a block of memory for storing `size` elements.
- Syntax to use `delete[]`: `delete[] pointer-variable;`
  - pointer-variable here points to a block of memory

// Pointer initialized with null pointer

// Dynamically allocates memory for 7 floating point numbers continuously of

// type double and returns pointer to the first element of the sequence, which

// is assigned to p. p[0] refers to first element, p[1] refers to second element ...

```
double *p = nullptr;
p = new double[7];
```

//graphically,

2.5	**-3.14**					
p[0]	p[1]	p[2]	p[3]	p[4]	p[5]	p[6]

```
p[0] = 2.5; p[1] = -3.14;
delete[] p; // release allocated memory
```

- Another graphic view:

Name	Type	Contents	Address
p	double array pointer	0x3D3B38	0x22FB66

Memory heap (free storage we can use)	
…	
0x3D3B38	2.5
0x3D3B39	-3.14
…	

Another example

```
Func()  /* C++ version dyn_array_CPP_ver.cpp , see also  zxu2/Public/dyn_array.c */
{
          double *ptr, a[100];
          ptr = new double[10];
          for(int i = 0; i < 10; i++)
              ptr[i] = -1.0*i;
          a[0] = *ptr;
          a[1] = *(ptr+1);   a[2] = *(ptr+2);
          delete[] ptr; //Must have if ptr is not needed anymore.
}
```

Normal array vs using new

- Normal arrays are allocated/deallocated by compiler (If array is local, then de-allocated when function returns or completes).
- Dynamically allocated arrays always remain there until either they are deallocated by programmer or program terminates.

Memory Management

- Common errors
  1. Leaked objects: Use new and then forget to delete the allocated object
  2. Premature deletion: delete an object that they have some other pointer to and later use that other pointer.
  3. Double deletion: an object is deleted twice.
  4. Go beyond range.

- Use debug option to compile and use debugger to step through the code.

  ~zxu2/Public/dyn_mem_alloc_CPP_ver.cpp

  ```
  icpc -g -std=c++11 dyn_mem_alloc_CPP_ver.cpp
  ```

  – See Appendix_4_Debugging-Serial-Parallel.pdf for using gdb debugger.

Write better code (exception safety, 1st try)

- Consider the situation that the program used all available memory and new returns zero (a null pointer) or throw an exception upon failure.

```
Func()  /* dyn_mem_alloc_CPP_ver.cpp, see also  zxu2/Public/dyn_array.c */
{
     double *ptr;
     ptr = new (std::nothrow) double[1000000ul];
     if(nullptr == ptr){
        cout<<"ERROR, memory allocation failed in Func()\n";
        exit(-1);
     }
     //… some codes use ptr
     delete[] ptr;
}
```

// If the system cannot allocate the memory, new will return nullptr
// rather than throwing std::bad_alloc

See also Sec. 3.1, 3.2 in Writing-clean-code.pdf

A very quick introduction of functions-
for purpose of writing dynamic memory allocation function

- A function is a named sequence of statements.

- they have
  - a name
  - a return type
  - at least 0 arguments

- The most common syntax to define a function is:

```
type name ( parameter1, parameter2, ...)
{ statements
}
```

For example: `int square(int x) { return x*x; }`

- use them to replace repetitive code, and to facilitate divide-and-conquer and abstraction and to write structured program.

example – `square()` function

```
int square(int x)  // indicates this defines a function
{
   return x*x;     // computes, returns the promised value
}
// Up to one value may be returned;
// it must be the same type as  the return type.
```

- `"int x"` is the parameter of the function. A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function.
- To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value.

```
#include <iostream>
#include <new>

int square(int); // declaring function prototype
                 // This makes code easier to read
                 // declaration of prototype can also
                 // be done in a header file (*.h)
using namespace std;

int main()
{
    int x;
    x = square(2);
}

int square(int x) { //definition
  return x*x;
}
```

Quick summary: declarations and definitions

- A function must be **declared** before it is used.
- The declaration is the function's prototype:
  - name
  - return type
  - parameter types

- The function **definition** is the code, in `{ }`, which defines what the function actually does
  - this includes any returning, etc

- can legally call it like:

```
auto a = square(2);
auto b = square(2*3*9);
```

but *not* like:

```
int v1 = square();         int v2 = square;
int v3 = square(1,2);
int v4 = square("two");    int v5 = square("2");
```

// Argument type matters.

- Argument order matters

```
int raiseToPower(int base, int  exponent) {
    int result = 1, i;
    for (i = 0; i < exponent; i = i + 1) {
        result = result * base;
    }
    return result;
}
```

- `raiseToPower(2,3) returns 8`
- `raiseToPower(3,2) returns 9`

Why bother?

- Reduces typing. Instead of full code every time needed, just call function
- Reduces logic errors. Make mistakes once instead of all the time
- Makes code more readable. A well-named function is more descriptive than a bunch of code.
- Makes testing easier. Know there is a problem earlier in development

A few technicalities

- The declaration of a function fully specifies what you need to call it.
  - name
  - parameters
  - return type
- The declaration must come before it is used. Good practice to declare function includes:
  - 1). Declare in a header file (e.g. *.h).
  - 2) Declare at the beginning of the source code file (E.g. *.cpp)
  - See example: files under ~zxu2/Public/src and ~zxu2/Public/include

Dynamic memory allocation in a function

To this end, we will be able to:

1. specify size of dynamic memory through parameters of the function.
2. Write safe code involving dynamic memory.

3. Demo: Public/dyn_array_CPP_ver.cpp
   - `double* alloc_double();`
   - `double* alloc_vec(int);`

4. dyn_array_from_func.cpp // this is more involved. Let's do later

An even better dynamic memory allocation function

Check memory allocation error. If occurred, let caller handle.

Technique detail: use the return value of the function to tell error or success.

Q: How to return memory address of the allocated dynamic memory from the function?

This will be addressed after discussing passing info to function.

A technicality

https://blogs.cisco.com/performance/the-mpi-c-bindings-what-happened-and-why

Memory Allocation/Free Functions in C

- `void   *malloc(size_t   number_of_bytes)`

  -- size_t is an unsigned integral data type (used to represent size of objects in bytes)

  -- This function allocate a contiguous portion of memory

  -- It returns a pointer of type void* that is the beginning place in memory of allocated portion of size number_of_bytes.

- `void free(void * ptr);`

  -- A block of memory previously allocated using a call to malloc, calloc or realloc is deallocated, making it available again for further allocations.

void type

- “`void`” type: was introduced to make C syntactically consistent. The main reason for void is to declare functions that have no return value. The word "`void`" is therefore used in the sense of "empty" rather than that of "invalid".

- A variable that is itself declared void is useless. Void pointers (type `void *`) are a different case, however.

- A void pointer is a generic pointer; any pointer can be cast to a void pointer and back without any loss of information. Any type of pointer can be assigned to (or compared with) a `void` pointer, without casting the pointer explicitly.

Demo code: dyn_mem_alloc.c

```
…
int main()
{
     int   i;
     double  a[100], *b = NULL,
             *bb = NULL, *c = NULL, *cc = NULL;

     bb = (double *)malloc(sizeof(double));
         /*allocate memory for storing one double */
    *bb = -4.5;

    b = (double *) malloc(array_size * sizeof(double));
             /* allocation in C*/
             /* allocate an array of double*/
             /*The memory holding this array is continuous */
…
}
```

C printf

```
int printf ( const char * format, ... );
```

Writes the C string pointed by format to the standard output (stdout). If format includes format specifiers (subsequences beginning with %), the additional arguments following format are formatted and inserted in the resulting string replacing their respective specifiers.

http://www.cplusplus.com/reference/cstdio/printf/

```
double *Func_C()
/* C version, allocate memory for a double */
{
     double *ptr;
     ptr = (double *)malloc(sizeof(double));
     *ptr = -2.5;
     return ptr;
}
```

```
double *Func_C_array()
/* C version, allocate memory for 100 doubles */
{
      double *ptr;
      ptr = (double *)malloc(100*sizeof(double));
      ptr[0] = -2.5;
      ptr[33] = -2.5;
      return ptr;
}
```

Write safe `double *Func_C();`
Write safe `double *Func_C_array();`

- What if malloc() (or `new`) failed?
- Base code: dyn_array.c

Migrate to C++, write code mixing C, C++

Change the header files

- The ".h" headers dump all their names into the global namespace, whereas the newer forms keep their names in namespace std. Therefore, the newer forms are the preferred forms for all uses except for C++ programs which are intended to be strictly compatible with C.

http://www.cantrip.org/cheaders.html

- Demo: dyn_mem_alloc.cc

Dynamic multi-dimensional array

```
Func()   /* allocate a contiguous memory which we can use for 20 ×30 matrix */
{     /*version 1, fix row dimension, allocate columns dynamically*/
      double *matrix[20];
      int    i, j;
      for(i = 0; i < 20; i++)
            matrix[i] = new double [30];
      //... Now matrix can be used.
      matrix[2][3] = 3.14;  //save to 3rd row, 4th column position
}
```

- Remark: Better make both rows and columns dynamically allocated.

Demo: second version

- Use dynamic array of pointers.
- `~zxu2/Public/dyn_2dmatrix.cpp`

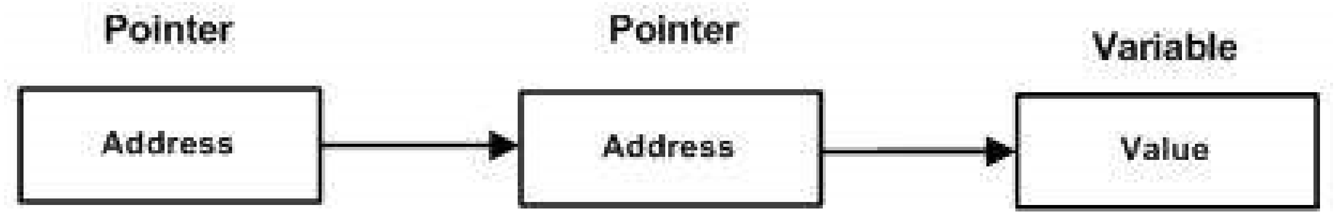
Third attempt of dynamic multi-dimensional array

```
Func()  /* allocate a contiguous memory which we can use for 20 ×30 matrix */
{ // Can you write a C version?
        double **matrix;
        int    i, j;

        matrix = new double* [20];
        matrix[0] = new double [20*30];

        for(i = 1; i < 20; i++)
            matrix[i] = matrix[i-1]+30;
        // now matrix can be used.
        for(i = 0; i < 20; i++)
        {
              for(j = 0; j < 30; j++)
                    matrix[i][j] =  (double)rand()/RAND_MAX;
        }
}
```

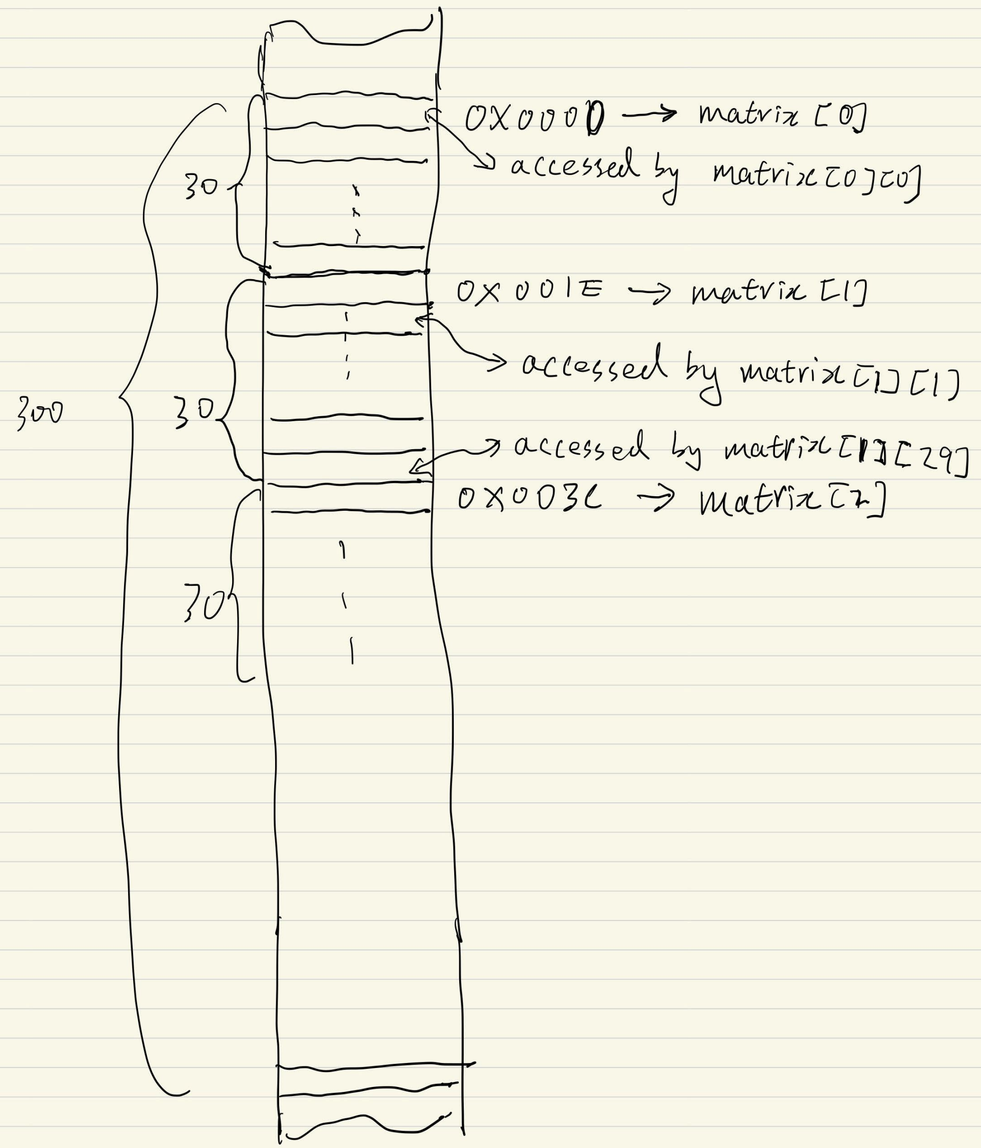
- Quiz: can you delete the memory? Integrate this function into the code.

Pointer to pointer

- https://www.tutorialspoint.com/cplusplus/cpp_pointer_to_pointer.htm



- `double **matrix;`
- `// A pointer-to-pointer (double pointer) is used`
- `// to store address of a pointer variable.`


0X0000 → matrix[0]
accessed by matrix[0][0]
0X001E → matrix[1]
accessed by matrix[1][1]
accessed by matrix[1][29]
0X003C → matrix[2]
30
30
30
300

Write better code

1. Pass dimensions of the matrix to the function
2. In the memory allocation function, check if memory allocation failed.
3. Return allocated memory so that others can use.

- In class demo: **Public/dyn_2dmatrix.cpp**.

Base code: ~zxu2/Public/sample_dyn_matrix

Q: Implement a function for allocate memory for dynamic 3D array? (try yourself before checking out `Public/dyn_mem_3D_array.cpp`, `Public/sample_dyn_matrix`)

More codes:

~zxu2/Public/dyn_mem_alloc_CPP_ver.cpp

Q: what would happen when uncommenting part of the code?

~zxu2/Public/dyn_array_from_func.cpp

Here another approach “pass by pointer” is used to pass information into a function.

~zxu2/Public/dyn_array.c

References

- A *reference* is an alias for an object (or variable). It is usually implemented to hold a machine address of an object (or variable), and does not impose performance overhead compared to pointers.
  - The notation **X&** means “reference to **X**”.

```
int    x;
int& ref = x;  // A reference must be initialized when it is created.
         //ref is an integer reference initialized to x
```

- Differences between reference and pointer.
  1. A reference can be accessed with exactly the same syntax as the name of an object.
  2. A reference always refers to the object to which it was initialized.
  3. There is no “null reference”, and we may assume that a reference refers to an object.

```
using namespace std;

int main () {
   // declare simple variables
   int    i;
   double d;

   // declare reference variables
   int&    r = i;
   double& s = d;

   i = 5;
   cout << "Value of i : " << i << endl;
   cout << "Value of i reference : " << r  << endl;

   d = 11.7;
   cout << "Value of d : " << d << endl;
   cout << "Value of d reference : " << s  << endl;

   return 0;
}
```

// check the code ~zxu2/Public/reference.cpp

```
void f()
{
     int var = 1;
     int& r{var}; // same as: int& r = var;
     int x = r;      // x becomes 1
     r = 2;             // var becomes 2
     ++r;              // var becomes 3
     int  *pp = &r;            // pp points to var.
}

void f1()
{
     int var = 1;
     int& r{var}; // r and var now refer to the same int
     int& r2;     // error: initialization missing
}
```

- Remark:
  We can not have a pointer to a reference.
  We can not define an array of references.

SEC 5. FUNCTIONS

A test

```
#include <iostream>
int nFive = 5;
int nSix = 6;
void SetToSix(int *pTempPtr);
using namespace std;

int main()
{
    int *pPtr = &nFive;
    cout << *pPtr;

    SetToSix(pPtr);
    cout << *pPtr;
    return 0;
}

// pTempPtr copies the value of pPtr!
// I.e., pTempPtr stores  the content of pPtr
void SetToSix(int *pTempPtr)
{
     pTempPtr = &nSix;

    cout << *pTempPtr;
}
```

//Public/ACMS40212/five_six.cpp

declarations and definitions (a repeat)

- A function must be **declared** before it is used.
- The declaration is the function's prototype:
  - name
  - return type
  - parameter types

- The function **definition** is the code, in {}, which defines what the function actually does
  - this includes any returning, etc

example prototypes

- void foo();
  - takes no arguments, no returned value
- int bar(int, int);
  - takes two int's, returns an int
- string getname();
  - returns a string, takes no arguments
- void modifyvec_pointer(int*,int);
  - takes a pointer to int, and possibly modifies it
- void modifyvec_ref(vector<int>&);
  - takes a reference to a vector<int>, and possibly modifies it

header and source

- Declarations go in header files
  - .hpp (or .h)
  - function prototypes, terminated with semicolons

- Definitions go in source files
  - .cpp (or .c)
  - function definitions, with curly braces and returning statements

- This is for compliance with the ODR – one definition rule

and now a demo

- what happens if we provide only the declaration?
- what happens if the declaration appears after the use of the function?
- what are the ways we can separate definition from declaration?
- Base code: ~zxu2/Public/sample_dyn_matrix
- ?

passing methods for functions

- pass by value
  - function gets a copy of the data fed in.
  - no ampersand
- pass by reference
  - function gets the original data itself. no copy. well and truly the same object
  - ampersand & after type name.
  - passed-in object is modifyable
- pass by const reference
  - well and truly same object, but const, so not modifyable
  - const&
- pass by pointer
  - function gets the address of the object. No copy
  - * after type name.
  - object referred to by the pointer is modifyable

examples of each

- `void foo(int v);` // by value, takes a copy into v
- `void foo(int& v);` // reference, v modifyable
- `void foo(const int& v);` // const reference, not // modifyable
- `void foo(int* v);` // pointer, modifyable

Passing arguments by value

- //see ~zxu2/Public/Func_arguments

```
#include<iostream>
void foo(int);

using namespace std;
void foo(int y){
     y = y+1;
     cout << "y + 1 = " << y << endl;
}

int main()
{
    foo(5); // first call
    int x = 6;
    foo(x); // second call
    foo(x+1); // third call

    return 0;
}
```

When foo() is called, variable y is created, and the value of 5, 6 or 7 is copied into y. Variable y is then destroyed when foo() ends.

Remark: Use debug option to compile the code and use debugger to step through the code.
icc -g pass_by_val.cpp

Pass by address (or pointer)

```
#include<iostream>
void foo2(int*);
using namespace std;

void foo2(int *pValue)
{
    *pValue = 6;
}

int main()
{
    int nValue = 5;

    cout << "nValue = " << nValue << endl;
    foo2(&nValue);
    cout << "nValue = " << nValue << endl;
    return 0;
}
```

Passing by address means passing the *address of the actual argument variable*. The function parameter must be a pointer. The function can then dereference the pointer to access or change the value being pointed to.

- 1. It allows us to have the function change the value of the argument.
- 2. Because a copy of the argument is not made, it is fast, even when used with large structures or classes.
- 3. Multiple values can be returned from a function.

Passing Arrays

Passing 1D array.

1. `int ProcessValues (int a[], int size);` // works with ANY 1D array. The compiler only needs to know that the parameter is an array; it doesn't need to know its size. Moreover, `ProcessValues()` receives access to the actual array, not a copy of the values in the array. Thus any changes made to the array within the function change the original array.
2. When declaring parameters to functions, declaring an array variable without a size is equivalent to declaring a pointer. Thus "`int ProcessValues (int *a, int size)`" is equivalent to the above declaration.

When we pass a 2D array to a function we must specify the number of columns -- the number of rows is irrelevant. We can do: `f(int a[][35]) {.....}`

Pass 3D array

```
int pass_3darray(int a[][5][3], int size);
```

`size:` this parameter tells the function the size of the first dimension of 3D array `a`.

When this function is called, the size of the 2nd and 3rd dimension of the 3D array must match with `5` and `3`, respectively.

Revisit codes

- `Public/dyn_array_from_func.cpp`
  - The address of dynamically allocated memory now is returned from argument of function.
  - What's good, what's better about this code?
- 2D array case: see function `my_alloc_double_matrix_better()` in `Public/sample_dyn_matrix`
- `f(int a[][35]) {.....}` is different from `f(int **a) {.....}.`

Pass by reference

//Public/Func_arguments/pass_by_ref.cpp

```
#include<iostream>
void foo3(int&);
using namespace std;

void foo3(int &y) // y is now a reference
{
    cout << "y = " << y << endl;
    y = 6;
    cout << "y = " << y << endl;
} // y is destroyed here

int main()
{
    int x = 5;
    cout << "x = " << x << endl;
    foo3(x);
    cout << "x = " << x << endl;
    return 0;
}
```

Demo

- Since a reference to a variable is treated exactly the same as the variable itself, any changes made to the reference are passed through to the actual argument.

- Now consider: `alloc_vec_cpp_ref(int,double*&)` in Public/dyn_array_CPP_ver.cpp

- How to write a reference version for dynamic allocation of 2D array?
- Base code: `Public/dyn_2dmatrix.cpp`, `Public/sample_dyn_matrix`.

SEC 6. ERRORS

Agenda

- Sources of Errors
- Compile-time
- Link-time
- Run-time
- Exceptions
- Estimation
- Debugging
- Pre- and post-conditions
- Testing

Where do errors come from?

- incomplete problem description
- incomplete implementation
- incorrect arguments to functions
- bad input to the program
    - from the user
    - from the internet
    - from another program
- unexpected state
- logic errors
    - failure to correctly implement

Compile time

- Syntax errors
  - those the compiler checks for,
  - this is what makes a source file (of any kind) parseable code.
- Type errors
  - trying to do incorrect things with a type.
  - this is the compiler being helpful – with error messages.
- ”Non-errors”
  - undesired narrowing conversions
  - many more

Link time

- When the parts of a program
  - object files, coming from .cpp files. one per.
  - other libraries
- fail to adhere to the rules.

- violation of the One Definition Rule
  - too many defns, or not enough. exactly one.
- linking to incompatible versions libraries

Run-time

- negative numbers when only positive numbers are expected.
- out-of-range errors, like accessing data that doesn’t exist.
- division by 0, for integers

- an infinite number of things can go wrong.

example

```
int area(int length, int width)
{
     return length*width;
}

int framed_area(int length, int width)
{
     return area(length-2, width-2);
}

int main()
{
     // use framed_area(...), area(...)
```

technique -- Caller deals with it

- Check for data validity before calling a function.

- example: function `area(x,y)` takes two `ints`. Put in negative `x` or `y`, get a negative area. Error.
- so, *before* calling area, the *caller* has to check for compatible argument values.

technique -- Callee deals with it

- Check for valid arguments inside called function.
- how to report the error? for now we will call `error()`
- example:

```
int framed_area(int x, int y)
{
        constexpr int frame_width {2};
        if (x-frame_width<=0 || y- frame_width<=0)
                error("non-positive area() argument called by
                        framed_area()");
        return area(x-frame_width,y-frame_width);
}
```

exceptions

- one of the things C++ has that C doesn't
- one of the contention things in life, exceptions.
- solves the problem of stack-propagating error notifications.

- what do you do to an exception? `throw` it!
- what else do you do to an exception? `catch` it.
- you throw when you are generating one,
- and you catch when you can.
- Demo: `~zxu2/Public/exception_demo.cpp`

C++ exception handling

- **throw** − A program throws an exception when a problem shows up. This is done using a **throw** keyword.
- **catch** − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.
- **try** − A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.
- Assuming a block will raise an exception, a method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch as follows −

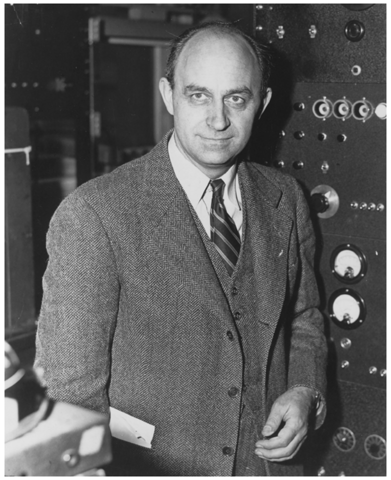
```
try {
   // protected code
} catch( ExceptionName e1 ) {
   // catch block
} catch( ExceptionName e2 ) {
   // catch block
} catch( ExceptionName eN ) {
   // catch block
}
```

how to know if a program is correct?

- ok, so how do we know if a program is doing what it is supposed to do?

- several techniques:

- estimation -- know approximately what is supposed to happen, by fermi estimation
- debugging -- use the debugger, and practices
- use pre- and post-conditions
- write unit tests



estimation

- knowing 'about what is supposed to happen'

- is your program giving ball-park correct answers?

- do you know a ball-park correct answer?
  - No? then why did you program. you can't effectively program until you have input and expected output...

debugging

- an art... here are some tips from me and our author

1. make your program easy to read. comment. choose good variable names. agree to use conventions
2. use print statements to display values of suspect variables
3. learn to use gdb, lldb, etc. the debugger.
4. sleep, take a break, ask a friend to look.

pre- and post- conditions

- The preconditions are the assumptions that the member function makes about its inputs (its parameters, if any, and the state of the object to which it is applied), and the postconditions describe the output when the function is done.
- comment the pre- and post-conditions for the function
- check for the condition.
- generate an error if condition violated.

- use that exception system to your advantage.
- document document document

testing

- The first and last word on program correctness – testing

- how do you know what to test? you did figure out some test cases and run them through pseudocode before programming, right? those are your tests!!!
- use an automated system.
  - a version control system. Git or Mercurial are hot right now
  - a testing library. Try Boost.UnitTest
  - automated build software. Try Jenkins (https://jenkins.io/).

SEC 7. GETTING STARTED

Agenda

- Strategies for getting started in a program

Suggestions 1

- What is the problem to be solved?
  - be very specific. sooooo very specific
  - Is the problem statement clear? clarify. find example input, and example output.
  - Define how you would like to interact with it

  - Question the manageability of the problem. Does it seem feasible? If not, choose something else, or hire more people

Suggestions 2

- Break the program into manageable parts

- Find standard components.
  - libraries and already-written and tested code solving sub problems.
  - pay attention to license restrictions
- Find ways to make the program be an assembly of parts
  - implement each object type separately
  - find ways to separate concerns

Suggestions 3

- Prototype
  - solve small limited versions of the subproblems.
  - experiment and try to break it
    - to bring about problems in our understanding, ideas, tools
    - to see if problem statement is clear
    - to find holes in our logic

- Then throw away the prototype, or keep only the best parts, and implement it again, better this time.

- Beware the sunk cost fallacy.
- Do not develop emotional attachment to code!

Sec 8. WRITING PROGRAM

Agenda

- Finding a (good) problem
- Thinking about the problem. The stages of development
- And then we do it.
  - choose a problem.
  - think about it.
  - program it.
  - ... test it ...
  - ???
  - profit

What's a good problem

- A good problem is well-stated.

- Well-defined input, and clearly specified output
- Limited scope. It is one problem, not a giant ball of problems.
- Is worth solving. Our problems are mostly worth solving to practice the patterns and learn the language
- Can be tested.

Ways to think about a problem

- The picture solves the problem.
  - draw draw draw
  - flow charts, example usage, example inputs and outputs.
- Role play. Be the data, and do what the data does.
- Research previous solutions and related problems.

Stages of development

- Analysis
  describe the problem.
  create requirements or specifications
- Design
  identify parts and components.
  brainstorm a working map of the parts and relationships
  find standard components solving subproblems
- Implementation
  write
  debug
  test
  maintain