C++ Session

C++ / Python Programming

Session #2

@Institut d'Astrophysique de Paris

Session Outline

- Part 1: C Reminder and Introduction to C++
- Part 2: Why Classes and Objects?
- Part 3: Virtual function and Exception
- Part 4: Templates

PART 1

C REMINDER AND INTRODUCTION TO C++

History

1. C with Classes (B. Stroustrup) – 1979
2. First edition – 1985
3. C++ 2.0 – 1989
4. C++98 (First ISO standard) – 1998
5. C++03 – 2003 – bug fix release for compiler writers
6. C++11 – 2011 – addition to libraries (threads, tuple etc.) and core language (auto, lambda etc.)
7. C++14 – 2014 – minor changes like function return type deduction
8. C++17 – 2017 – Major changes (std::string_view, std::any, a filesystem library based on boost)
9. C++20 – Upcoming

Advantages and Disadvantages

Advantages	Disadvantages
•Powerful •Fast •Available on most platforms •Widely used •Lots of libraries •Domains •Lots of mathematic libraries •Good for embedded systems •Good for simulations	•Memory management like pointers, allocation and deallocation •Debugging application is complex •Can’t support garbage collection

OUTLINE

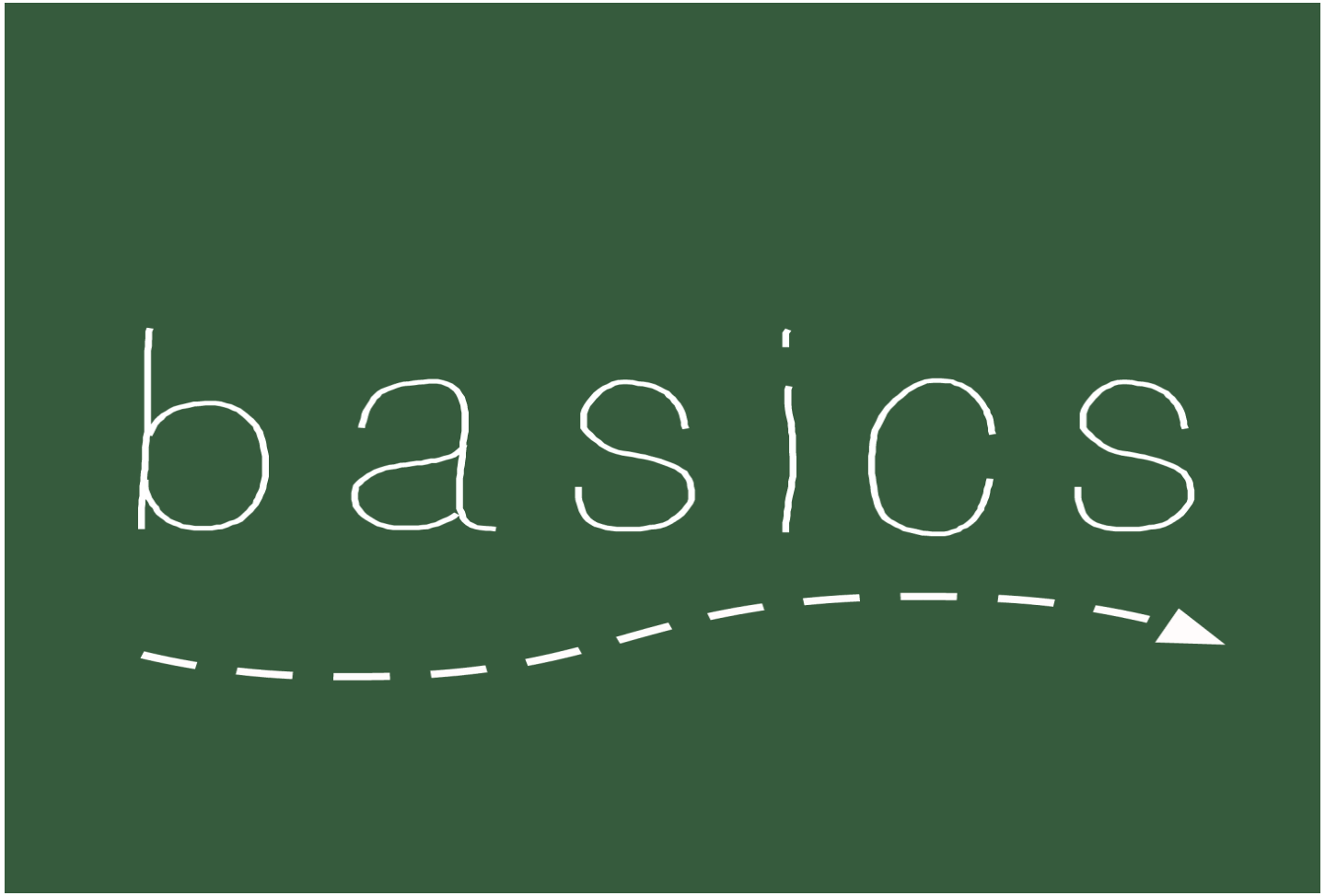
1. Introduction
   a. C++ Variables
   b. Variable declaration
   c. How to make a Program
   d. Comments
   e. Compilation and Execution
   f. Input / Output
2. If-Else
3. Switch Statement
4. Loops
5. Arrays
6. Pointers

OUTLINE

6. Pointers Vs Arrays
7. Pointers Vs References
8. Functions
   a. **return** Keyword
   b. **main** Function
   c. Scope of variable
   c. Function call by value
   d. Function call by Reference
   e. Function Overloading
   e. Arrays and Functions
9. Memory Allocation
10. Use of Pointers for Dynamic memory

OUTLINE

11. Memory Organization
12. Namespaces
13. Conclusion


basics

PURPOSE

1. What are variables and how to declare them?
2. How program executes?
3. How to handle input and output?
4. Use if-else condition and loops
5. What is an array? How it works?
6. What are pointers? Why we should use pointers? When and where to use pointers?
7. What are functions? How to call functions?
8. What is the use of namespaces?

C++ Variables

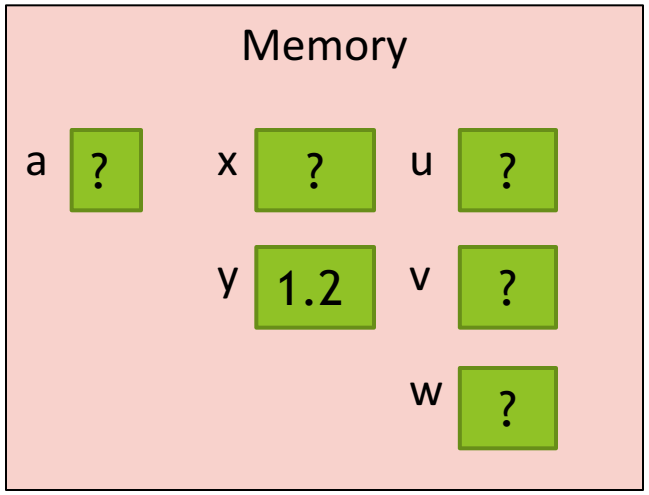
- C++ Identifiers
  - Keywords/reserved words vs. Identifiers
  - Case-sensitivity and validity of identifiers
  - Meaningful names!
- Variables
  - A memory location to store data for a program
  - Must declare all data before to use them in a program
- Type
  - define the information stored in variable (range, nature, size..) and also define the operations allowed to perform.

Variable declaration

- Type variable = Expression;
  - int a;
  - float x, y(1.2);
  - float u, v, w;

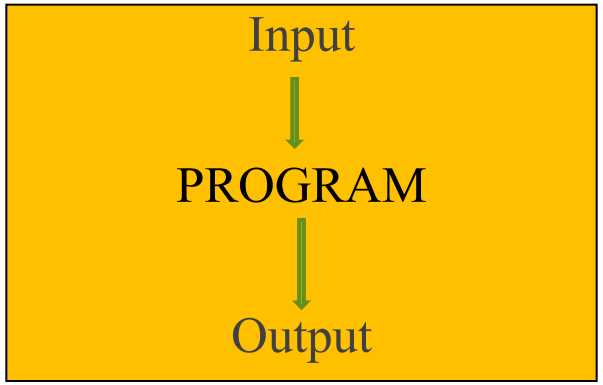
TYPE:

1. int
2. float
3. double
4. char
5. bool



How to make a program

USER POINT OF VIEW



There is no universal way to produce a program but you may have to follow some rules.

1. Specification
2. Identify Input
3. Identify Output

Specifications – need to be simplified and should be in sequence like a recipe.

Sequence:

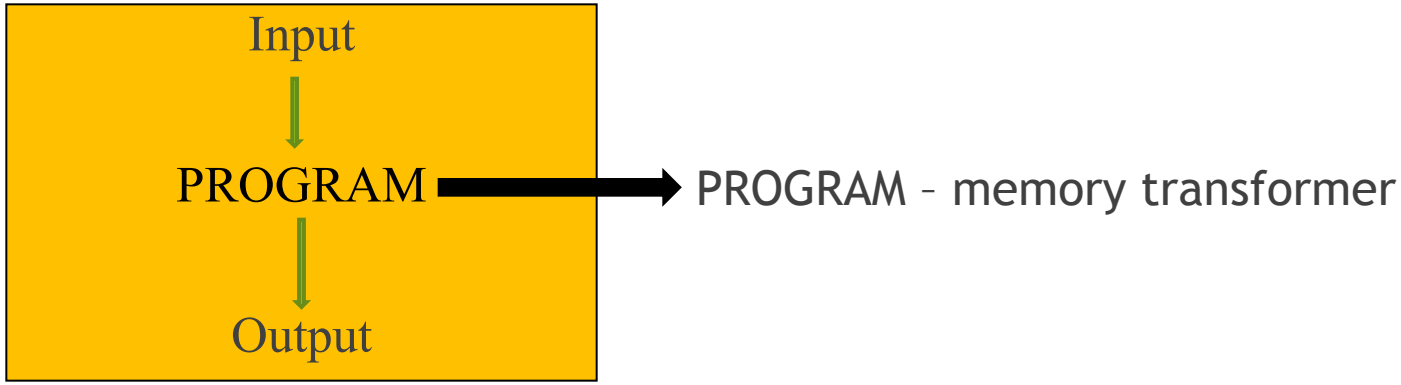
```
{
    statement_1;
    statement_2;
    ……..
    statement_n;
}
```

First Program:

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.  int i = 2, j = 5, t;
6.  cout << " i = " << i << ", j = " << j << endl;
7.  t = i;
8.  i = j;
9.  j = t;
10. cout << " i = " << i << ", j = " << j << endl;
11. return 0;
12.}
```

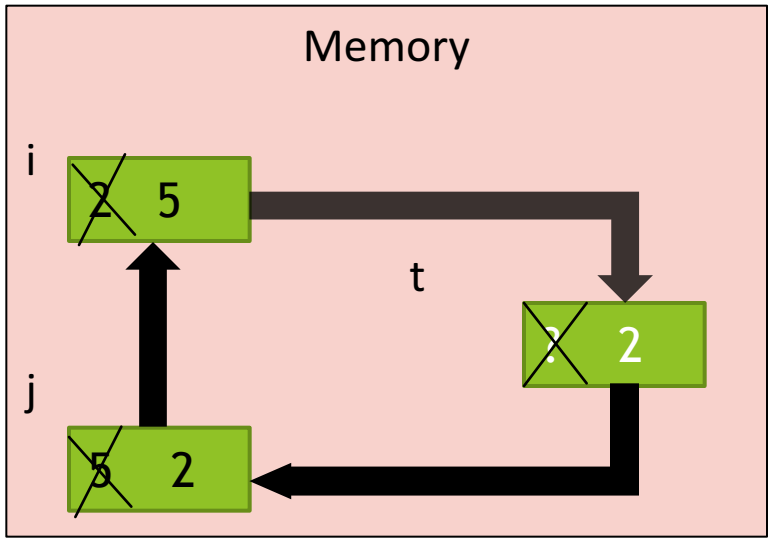
cont..

USER POINT OF VIEW



Example: Swapping

```
int i = 2, j= 5, t;
{
  t = i;
  i = j;
  j = t;
}
```



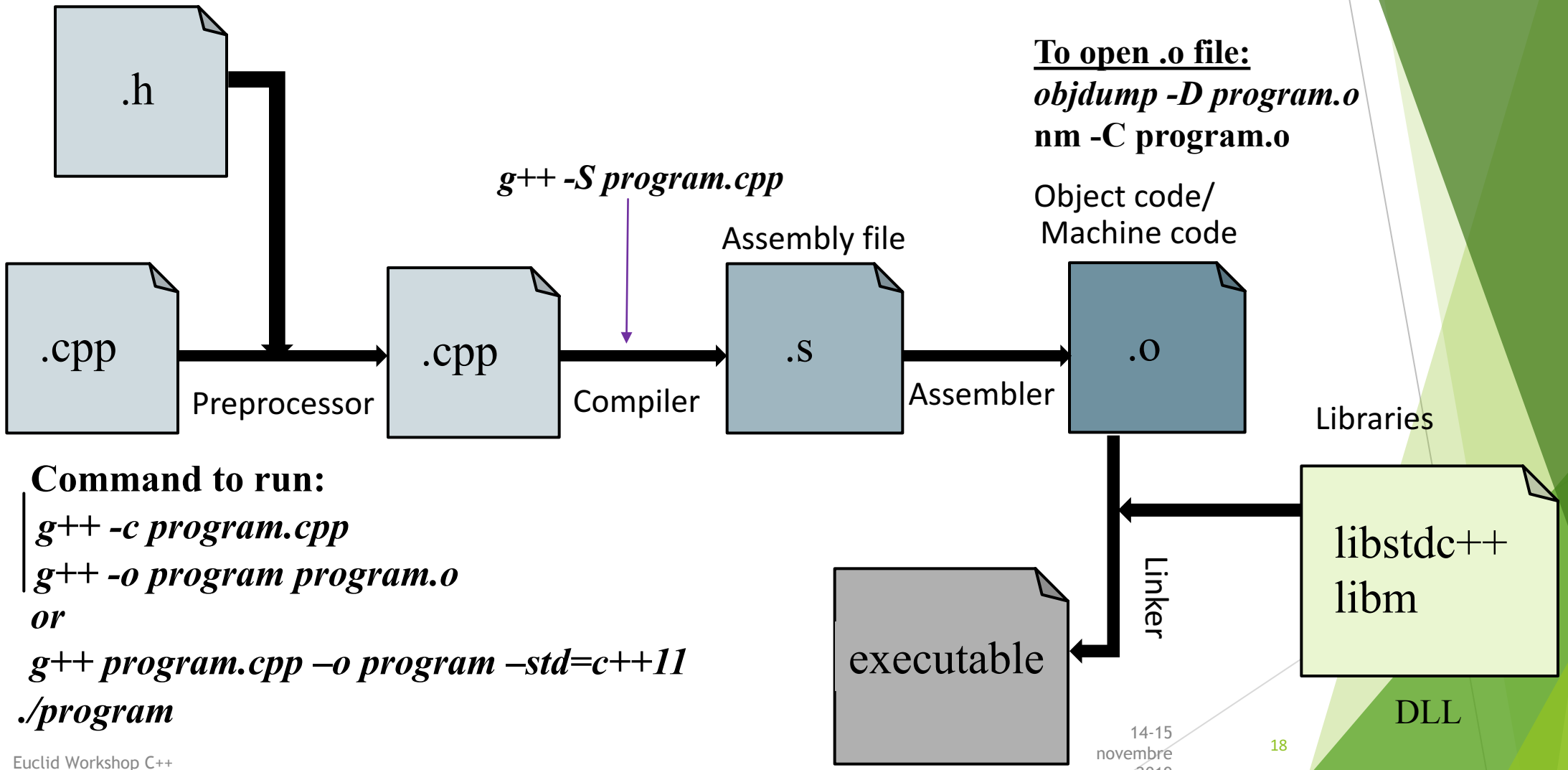
Comments

- Two styles of comments provided:
  - Comment starts with // and proceeds to end of line.
    // This is an example of a comment
  - Comment starts with /* and proceeds to first */
    /* this is an example of comment */
    "OR"
    /*This is an example of comment
    with multiple lines */
- DO NOT NEST - Comments of /* ··· */ style
  /*
  /* This line is part of a comment. */
  This is not part of any comment and will probably cause a compile error.
  */

First Program:

```
1. /* Program to swap the values of two variables */
2. #include <iostream>
3. using namespace std;
4. int main()
5. {
6.  int i = 2, j = 5, t;
7.  cout << " i = " << i << ", j = " << j << endl;
8.  t = i; //use of temporaire variable
9.  i = j;
10. j = t;
11. cout << " i = " << i << ", j = " << j << endl;
12. return 0;
13.}
```

Program Compilation & Execution



Command to run:

g++ -c program.cpp
g++ -o program program.o
or
g++ program.cpp –o program –std=c++11
./program

Common g++ command line options

OPTION	DESCRIPTION
-c	compile only (i.e., do not link)
-o file	use file for output
-g	include debugging information
-O***n***	set optimization level to n (0 almost none; 3 full)
-std=c++11	conform to C++11 standard
-pthread	enable concurrency support (via pthreads library)
-I***dir***	specify additional directory *dir* to search for include files
-L***dir***	specify additional directory *dir* to search for libraries
-l***lib***	link with library *lib*
-Wall	enable most warning messages
-Wextra	enable some extra warning messages not enabled by *-Wall*
-Werror	treat all warnings as errors

Input / Output

- Standard Input (cin)
- Standard Output (cout)
- Standard Error (cerr)
- Standard Log (clog)

```
#include <iostream>
using namespace std;
int main()
{
  int a, b;
  cout<< "enter the value of a:  ";
  cin>>a;
  cout<<"enter the value of b:  ";
  cin>>b;
  int sum = 0;
  sum = a+b;
  cout<<"sum of a and b is "<<sum<<endl;
  return 0;
}
```

If-Else

1. If (condition expression)
   statement;
2. If (condition)
   statement_1;
   else
   statement_2;

```
#include <iostream>
using namespace std;
int main()
{
  int x = 9;
  int y = 5;
  if (x > y) {
    cout << "x is greater than y";
  }
  return 0;
}
```

Conditional Expression:
1. Relational operators (>,<,==, <=,>=,!=)
2. Logical Operators (&&, ||, !)

Exercise

Make a program to convert Cartesian coordinates into Spherical coordinates.
(User Input: Cartesian coordinates (x, y, z)

$$r = \sqrt{x^2 + y^2 + z^2}$$

$$\varphi = \arctan \frac{y}{x}$$

$$\theta = \arccos \frac{z}{\sqrt{x^2 + y^2 + z^2}} = \arccos \frac{z}{r}$$

Switch statement:

- allows to select one of many code blocks to be executed.
- Syntax has form:

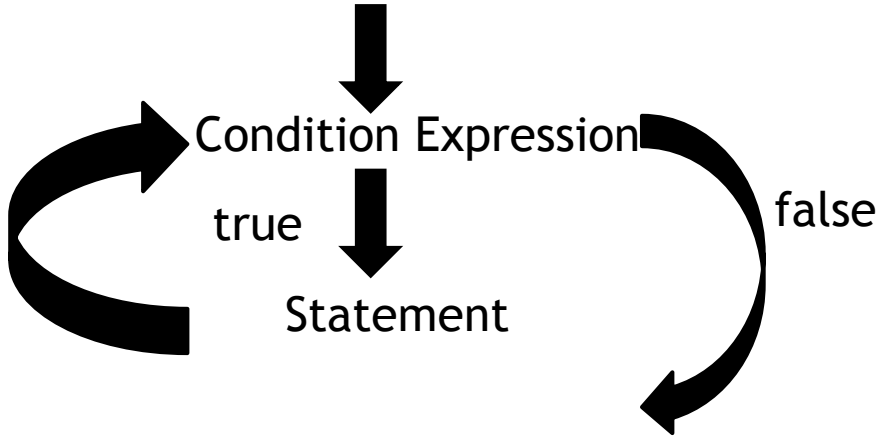
```
switch(expression) {
  case const_expr1:
    // code block/Statement
    break;
  case  const_expr2:
    // code block/Statement
    break;
  default:
    // code block
}
```

- The break and default keywords are optional

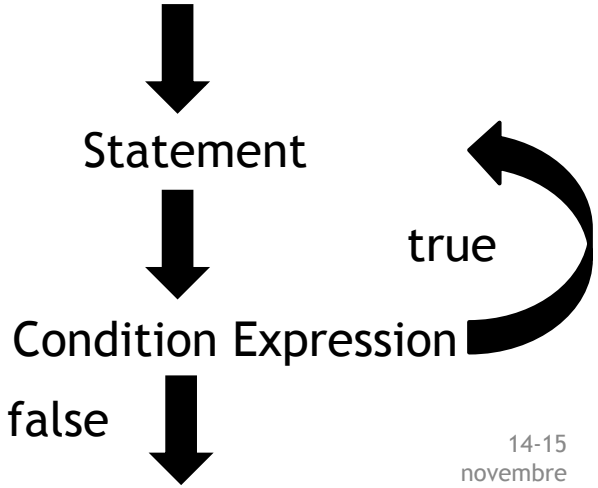
```
#include <iostream>
using namespace std;
int main() {
 int day = 4;
 switch (day) {
  case 1:
    cout << "Monday";
    break;
  case 2:
    cout << "Tuesday";
    break;
  case 3:
    cout << "Wednesday";
    break;
  case 4:
    cout << "Thursday";
    break;
  case 5:
    cout << "Friday";
    break;
  case 6:
    cout << "Saturday";
   break;
  case 7:
    cout << "Sunday";
   break;
  }
  return 0;
}
```

Loops

```
{
 Initialization;
 while( Condition ) {
   Statement
   Incrementation ;
 }
}
```




```
do
 statement;
while( Condition ) ;
```



Loops

for (Initialization ; Condition ; Incrementation) **//which is C representation**

Statement;

“OR”

for (Declaration ; Condition ; Incrementation)

Statement;

Incrementation:
1. Increment (++)
2. Decrement (--)

- Old

```
for (int i = 0; i < n; i++)
```

- New

```
for (int i(0); i != n; ++i)
```

Loops

// Sum of n numbers using for (int i = 0; i < n; i++)

```
1.  #include <iostream>
2.  using namespace std;
3.  int main() {
4.    int i, n, sum =0;
5.    std::cout << "enter the n number: ";
6.    std::cin >> n;
7.    for (int i=0; i < n ; i++) {
8.      sum = sum + i;
9.    }
10.   std::cout << "sum of first "<< n <<" numbers
      is: "<< sum << std::endl;
11.   return 0;
12. }
```

// Sum of n numbers using for (int i(0); i != n; ++i)

```
1.  #include <iostream>
2.  using namespace std;
3.  int main() {
4.    int i, n, sum =0;
5.    std::cout << "enter the n number: ";
6.    std::cin >> n;
7.    for (int i(0); i !=n ; ++i) {
8.      sum = sum + i;
9.    }
10.   std::cout << "sum of first "<< n <<" numbers
      is: "<< sum << std::endl;
11.   return 0;
12. }
```

Prefix and Postfix Increment

- In the **prefix version** (i.e., ++i), the value of i is incremented, and the value of the expression is the **new** value of i.
- Example:

```
int i = 10; // (1)
int j = ++i; // (2)
```

Here, first i is incremented to 11. Then, The **new** value of i is copied into j. So j now equals 11.

- In the **postfix version** (i.e., i++), the value of i is incremented, but the value of the expression is the original value of i.
- Example:

```
int i = 10; // (1)
int j =  i++; // (2)
```

The original value of i (which is 10) is copied into j, then i is incremented to 11. So j is equals to 10.

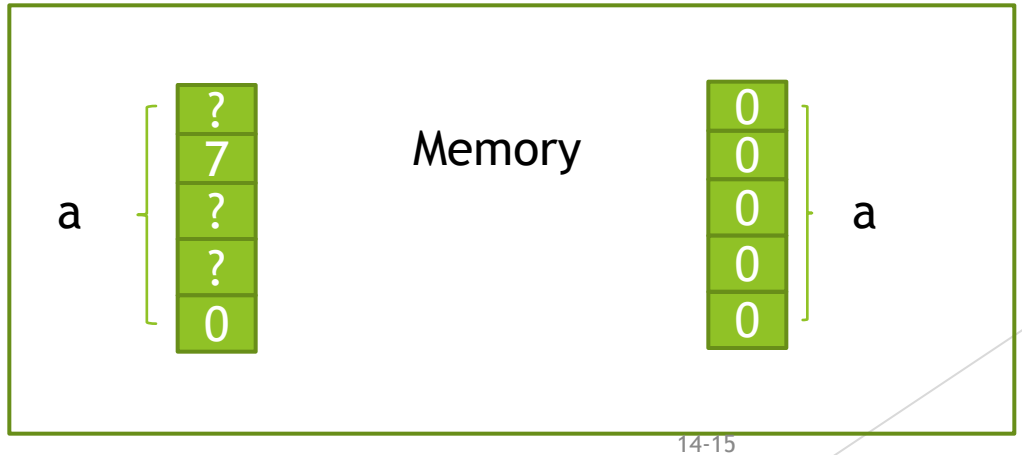
Arrays

- One name and several values. ***Type identifier [Size];***
- Array elements have unique index
- First element set at index 0

NOTE: Program doesn't check if index is bounds to array

Example:

```
int a[5];
for (int i=0; i<5;i++)
   a[i] = 0;
```



Exercise

Make a program to get average of the values stored in an integer type array.

(1. Delare an integer type array

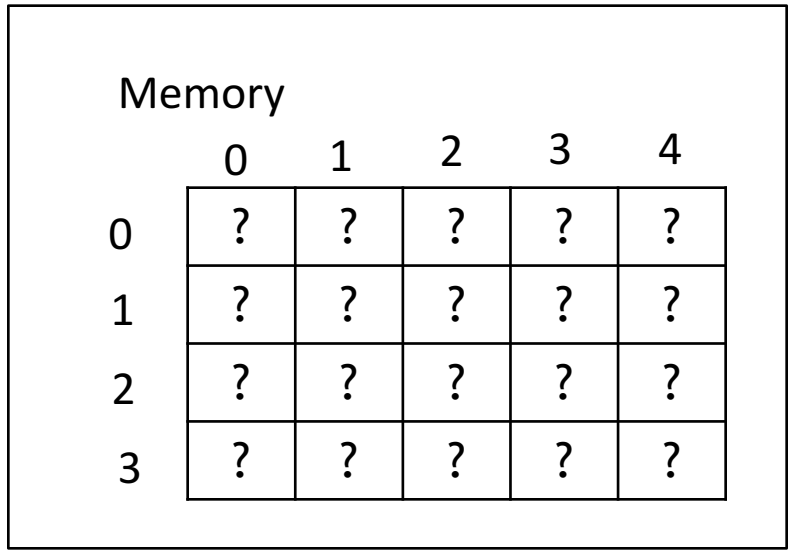
2. number of elements (user input)

3. store values in array (user input)

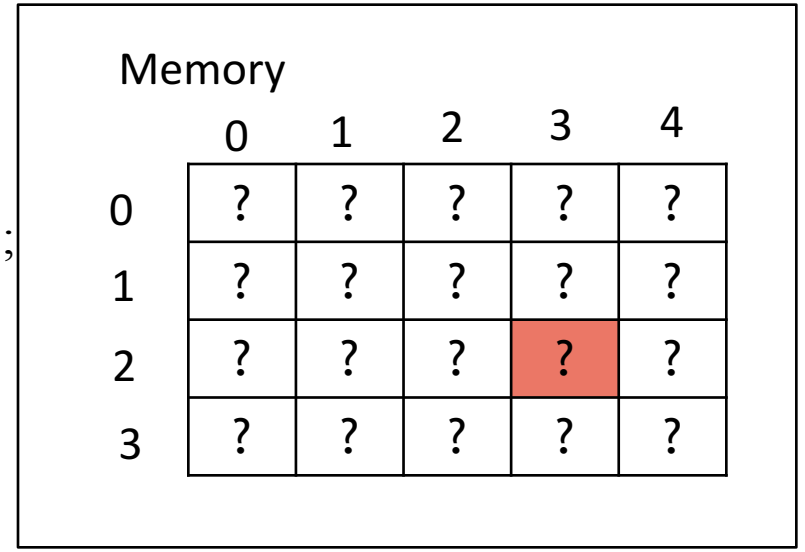
4. calculate average and print the value)

Multi-dimensional Array

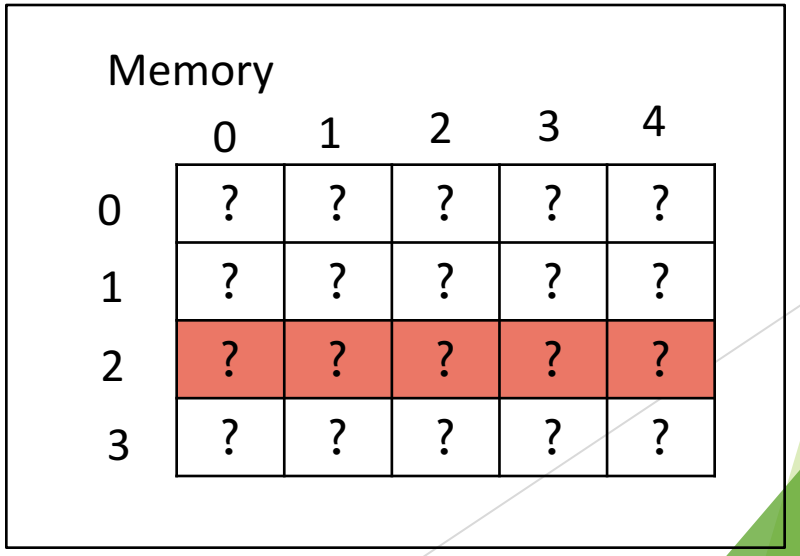
- ***Type identifier [Size_1] [Size_1] ;***
- Example: int array [4] [5];



array [2] [3];



array [2];



Exercise

Make a program to multiply two matrices and print the output.

{ 1. rows and columns for matrix1 and matrix2 (user input)

2. store values in matrices (user input)

3. perform multiplication and print the value }

Pointers (* and &)

- variable that contains reference of another variable
- <variable_type> *<name>;
- & -> stores address of the variable that points towards that variable

```
int x;
cout<< &x << endl;
```
⟶ this will print the memory address of variable x

- * -> “Contents of” binder at address of variable

```
int i = 5;
char *c = “Hello World!”;
int *j = &i;
```

#0	i = 5		H	e
#4	l	l	o	
#8	W	o	r	l
#12	d	!	\0	\0
#16	c #2		j #0	

cont..

We want pointer, that should points towards x address.

- int x, *i;
- i=x; //Wrong!
- *i=&x; // Wrong! *i is the value pointed by address whereas, &x is an address.
- i=&x; // Correct!
- *i=x; // Correct! *i is the value pointed by address and, x is a value.

Pointers VS Arrays

- Similar: Both are interchangeable (not always). For example, a pointer that points to the beginning of an array can access that array by using either pointer arithmetic or array-style indexing.

```
#include <iostream>
using namespace std;
const int MAX = 3;
int main () {
  int var[MAX] = {10, 100, 200};
  int *ptr;      // let us have array address in pointer.
  ptr = var;
  for (int i = 0; i < MAX; i++) {
    cout << "Address of var[" << i << "] = ";
    cout << ptr << endl;
    cout << "Value of var[" << i << "] = ";
    cout << *ptr << endl;      // point to the next location
    ptr++;
  }
  return 0;
}
```

Pointers VS Arrays

- Differences:
  1. Sizeof() Operator:
     a. sizeof(array) returns the amount of memory used by all elements in array
     b. sizeof(pointer) only returns the amount of memory used by the pointer variable itself
  2. & Operator:
     a. &array is an alias for &array[0] and returns the address of the first element in array
     b. &pointer returns the address of pointer
  3. a string literal initialization of a character array:
     a. char array[] = “abc” sets the first four elements in array to ‘a’, ‘b’, ‘c’, and ‘\0’
     b. char *pointer = “abc” sets pointer to the address of the “abc” string (which may be stored in read-only memory and thus unchangeable)

Pointers VS References

- Similar: both can be used to refer to some other entity (e.g., object or function)
- Differences:
  1. reference must refer to something, while pointer can have null value (**nullptr**)
  2. references cannot be rebound, while pointers can be changed to point to different entity
- use of pointers often implies need for memory management (i.e., memory allocation, deallocation, etc.), and memory management can introduce numerous kinds of bugs when done incorrectly
- often faced with decision of using pointer or reference in code but generally advisable to prefer use of references over use of pointers unless compelling reason to do otherwise, such as:
  1. must be able to handle case of referring to nothing
  2. must be able to change entity being referred to

Functions

- To avoid Repetition
- Speed up
- decrease number of bugs
- type – type of result
- Param – list and type of inputs
- statement sequence – Implementation code

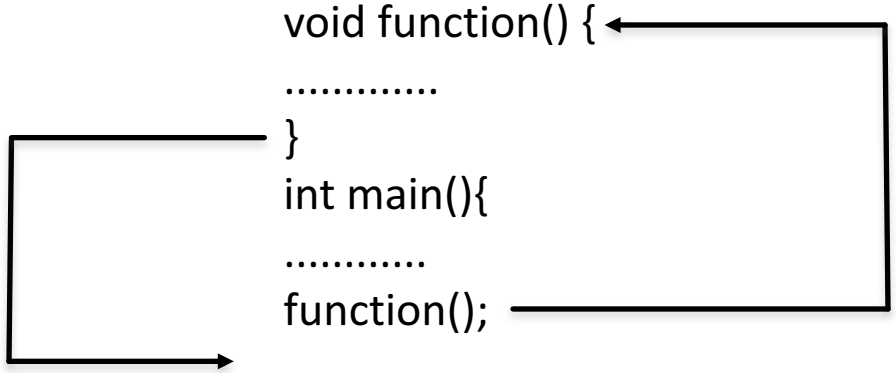
type identifier (Param1, Param2,.....)
Statement sequence

Param: Type identifier

Statement: *return* Expression

Note: Void type doesn't return any value

A function may be used as an expression or statement.
It is defined by its return value which function pass back to program.



Return keyword

- **return** statement used to exit function, passing specified return value (if any) back to caller
- code in function executes until **return** statement is reached or execution falls off end of function
- if function return type is not **void**, **return** statement takes single parameter indicating value to be returned
- if function return type is **void**, function does not return any value and **return** statement takes no parameter

The main Function

- entry point to program is always function called main
- has return type of **int** can be declared to take either no arguments or two arguments as follows:
  int main();
  int main(**int** argc, **char*** argv[]);
- two-argument variant allows arbitrary number of C-style strings to be passed to program from environment in which program run
- argc: number of C-style strings provided to program
- argv: array of pointers to C-style strings
- argv[0] is name by which program invoked
- argv[argc] is guaranteed to be 0 (i.e., null pointer)
- argv[1], argv[2], . . ., argv[argc - 1] typically correspond to command line options

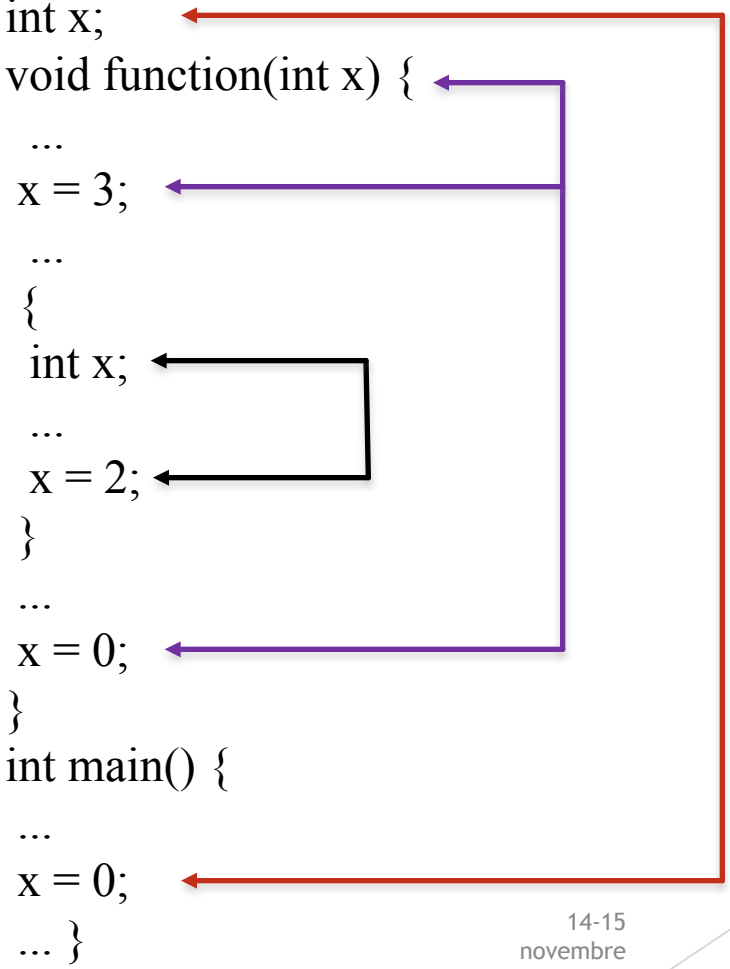
The main Function

- suppose that following command line given to shell:
  program one two three
- main function would be invoked as follows:
  int argc = 4; **char*** argv[] = {
  "program", "one", "two", "three", 0 };
- main(argc, argv);
  return value of main typically passed back to operating system
- can also use function **void** exit(**int**) to terminate program, passing integer return value back to operating system
- return statement in main is optional
  if control reaches end of main without encountering return statement,
- effect is that of executing "**return** 0;"

Scope of a variable

- Global variable: from all functions
- Local variable, parameter: inside their function
- Block variable: inside the brace
- arguments: inside parenthesis

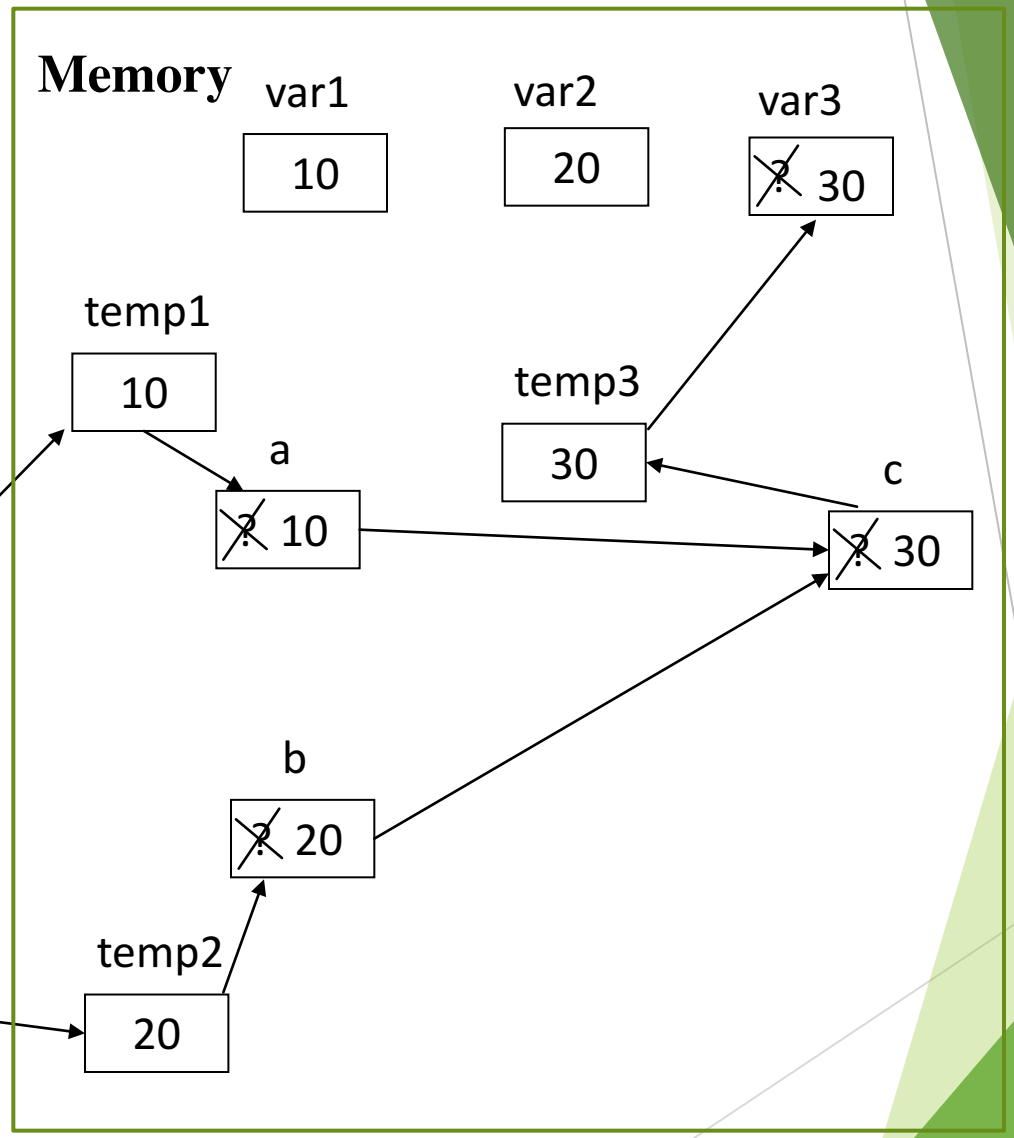
Example:



Function call by value

- Can not change the value of arguments in the calling function, it just use copies of the arguments.
- arguments can be expression or constants

```
int sum(int a, int b)
{
 int c=a+b;
 return c;
}
int main( )
{
 int var1 =10;
 int var2 = 20;
 int var3 = sum(var1, var2);
 cout<<“sum: ”<<var3<<endl;
 return 0;
}
```



Function call by reference

- function passes a reference to an object as an argument and access object directly and modify it.

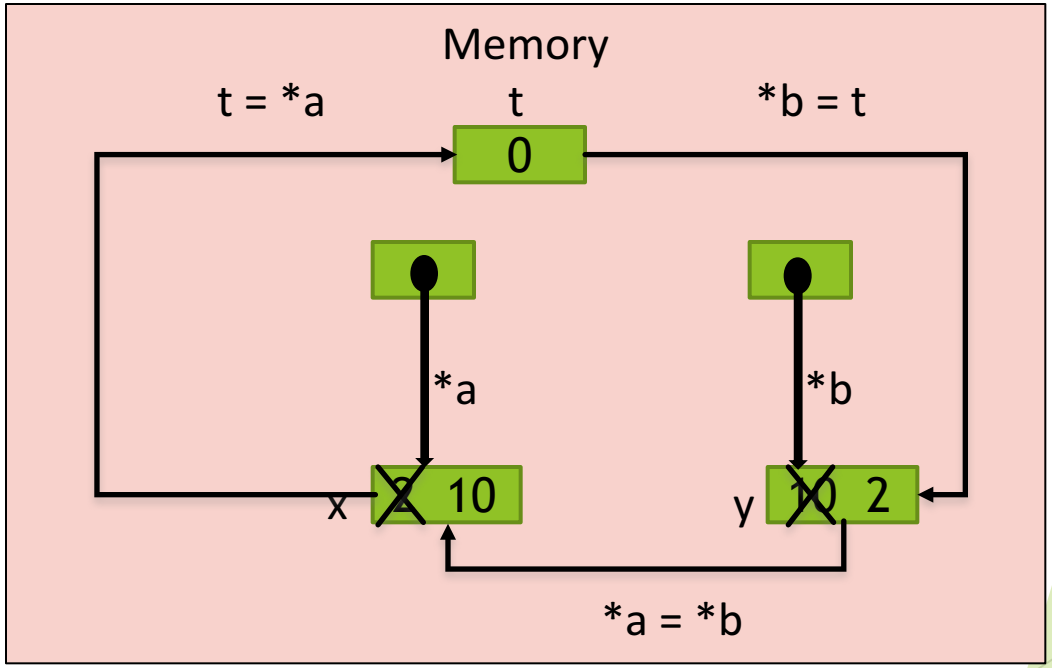
Parameter: Type * Identifier

Parameter Usage: * Identifier

Argument: & Identifier

Example:

```
void swap(int *a, int *b) {
  int t = *a;
  *a = *b;
  *b = t;
}
main() {
 int x = 2, y = 10;
 swap(&x, &y);
}
```



Note: We should not use this programming for swap function as it is already present in the <algorithm>.

Function exercise

Test the following functions:

```
void func1(int a){
    a += 2;
}
```

```
void func2(int *a){
    *a += 2;
}
```

```
void func3(int &a){
a += 2;
}
```

Do you find the expected results ?

Function exercise: solution

```
#include <iostream>

using namespace std;

void func1(int a){
        a += 2;
}

void func2(int *a){
        *a += 2;
}

void func3(int &a){
a += 2;
}
```

```
int main()
{
  int myA = 3;

  cout << "myA = " << myA << endl;

  func1(myA);
  cout << "after call func1, myA = " << myA << endl;

  func2(&myA);
  cout << "after call func2, myA = " << myA << endl;

  func3(myA);
  cout << "after call func2, myA = " << myA << endl;

  return 0;
}
```

```
$ ./func
myA = 3
after call func1, myA = 3
after call func2, myA = 5
after call func3, myA = 7
```

Function Overloading

- multiple functions can have same name as long as they differ in number/type of their arguments
- Example:

```
void print(int x) {
  std::cout << "int has value " << x << '\n';
}
void print(double x) {
  std::cout << "double has value " << x << '\n';
}
void main() {
  int i = 5;
  double d = 1.414;
  std::cout<<print(i)<<std::endl; // calls print(int)
  std::cout<< print(d) <<std::endl; // calls print(double)
  std::cout<< print(42) <<std::endl; // calls print(int)
  std::cout<< print(3.14) <<std::endl; // calls print(double)
}
```

Array & Functions

- Functions can not return an array value.
- Arrays, as parameters, are implicitly passed by address as pointer.

Memory Allocation: new and delete

- to allocate memory, use **new** statement and to deallocate memory allocated with **new** statement, use **delete** statement
(similar to malloc and free)

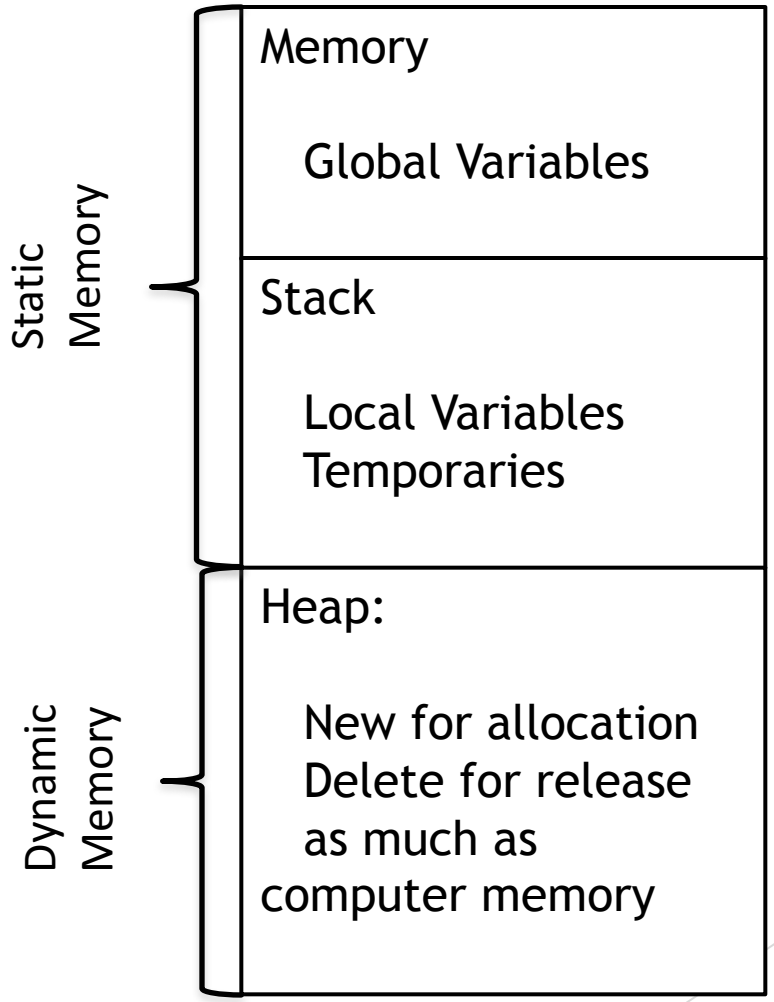
Use of pointers for Dynamic memory

Example 1: Allocation of an integer

```
int *i = new int;
int s = 0;
for (*i = 0; *i<10; (*i)++) {
  s = s + *i;
}
delete i;
```

Example 2: use of array

```
int n;
cout<<“Enter the required number:”
cin>>n;
 int *temp = new int[n];
for (int i = 0; i<n; i++) {
  cin>>temp[i];
  ……
}
delete [] temp;
```



Memory Organization

Global Variables

- Constructed when program start
- Size is fixed by compilation
- Destructed when program ends

Local Variables

- Constructed when function calls
- Size is fixed in function call
- Destructed when function returns

Heap Data

- Constructed when ***new*** calls
- Size is fixed by application
- Destructed when ***delete*** calls

Flexible but need to manage pointers

Namespaces

- mechanism for reducing likelihood of naming conflicts (i.e., attempt to use same identifier to have different meaning in various places in code)
- has general syntax:
  namespace *name* {
  code
  }
- identifiers only have to be unique within a single namespace same identifier can be re-used in different namespaces
- scope-resolution operator (i.e., ::) used to specify namespace to which particular identifier belongs
- **using** statement can be used to make identifiers declared in different namespaces appear as if they were in current namespace

Namespaces Example

```
1.  #include <iostream>
2.  using namespace std;

3.  // first name space
4.  namespace first_space {
5.    void func() {
6.      cout << "Inside first_space" << endl;
7.    }
8.  }

9.  // second name space
10. namespace second_space
11.   void func() {
12.     cout << "Inside second_space" << endl;
13.   }
14. }
```

```
1.  int main () {

2.    // Calls function from first name space.
3.    first_space::func();

4.    // Calls function from second name space.
5.    second_space::func();

6.    return 0;
7.  }
```

Conclusion

- C++ is a derivation of C
- Pointers are powerful so think about pointers
  ....Complicated but required
- Start some simple C++ programs and just program it.