Lecture 1 - **Basic C++**

An old friend with new powers……

Lecture Overview

- Introduction to C++
  - Control Statements
  - Declarations
  - Memory allocation & deallocation
  - Functions
  - Useful C Library in C++

What is C++?

- Developed by **Bjarne Stroustrup**
  - First commercial release in 1985
  - Originally known as "**C with Classes**"
  - Renamed to "**C++**" in 1983
  - C++ > C
- Main features:
  - General purpose
  - Object Oriented
  - Compatibility with C
    - More on this later…

The Good and Bad News

- **Good News:**
  - Only minor incompatibility with C
    - Most programs introduced in CS1010/E is valid and compilable
  - Proficiency in C++ is a great advantage:
    - Much sought after in the industry
    - Picking up other OO languages like Java, C# is relatively easy
- **Bad News:**
  - It is a HUGE and COMPLEX language
  - Compatibility with C detracts from pure Object Oriented approach

Advice

- Unlike CS1010/E, we are **not** concentrating on the programming language itself
  - It is a "vehicle" to discuss and implement data structures and algorithms
- CS1020E is more **conceptual based** and “higher level”
  - Ideas that are true regardless of the actual implementation language
- However, more than 30% of your CA comes from actual hands-on:
  - Labs: 20%, PE: 15%
  - Programming based questions in midterm and finals
- Conclusion:
  - Try **HARD** to be familiar with C++ in the first few weeks

Simple C++ Program

Getting Started

Input and Output

- Output using **`cout`**
- Input using **`cin`**
- To use either **`cin`** or **`cout`**, add the following two lines to the start of program

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

- Do not be alarmed of the above
  - Full explanation will be given later
  - At this point, just "cut and paste" into every C++ program ☺

“Hello World!” in C and C++

```
#include <stdio.h>


int main( ) {

  printf (“Hello World!\n");

  return 0;
}
```

C version

```
#include <iostream>

using namespace std;

int main( ) {

  cout << “Hello World!” << endl;

  return 0;
}
```

C++ version

Notes on C++ lectures

- Assume you have prior **C** programming knowledge
- "Gentle" introduction to C++:
  - Start by revision of C constructs
  - Minor additions are introduced first
  - Major and hard to understand topics later
- Topics are tagged:
  - [new] : topics introduced in C++, may not valid in C
  - [expanded] : topics covered in C, but greatly expanded in depth
- Topics without tags are revision on basic language constructs valid in both C and C++

Control Statements

Program Execution Flow

Approximating PI: **A Quick Test**

- Instead of going through the basic control statement, let's solve a simple problem
  - If you can do it easily, then your understanding of the basic control statements are largely intact ☺

- One way to calculate the PI π constant:

$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \dots\dots\dots$$

- Write a program to:
  - Ask user for number of terms to be used
  - Calculate the approximation and output

Selection Statements [For Reading]

```
if (a > b) {
    ...
} else {
    ...
}
```

- **if-else** statement
- Valid conditions:
  - Comparison
  - Integer values (0 = **false**, others = **true**)

```
switch (a) {
    case 1:
        ...
        break;
    case 2:
    case 3:
        ...
    default:
}
```

- **switch-case** statement
- Variables in **switch( )** must be integer type (or can be converted to integer)
- **break** : stop the fall through execution
- **default** : catch all unmatched cases

Repetition Statements **[For Reading]**

```
while (a > b) {
    ... //body
}
```

```
do {
    ... //body
} while (a > b);
```

- Valid conditions:
  - Comparison
  - Integer values (0 = false, others = true)
- `while` : check condition before executing body
- `do-while`: execute body before condition checking

```
for (A; B; C) {
    ... //body
}
```

- `A` : initialization (e.g. `i = 0`)
- `B` : condition (e.g. `i < 10`)
- `C` : update (e.g. `i++`)
- Any of the above can be empty
- Execution order:
  - `A, B, body, C, B, body, C` …

Declaration

Simple and composite data types

Simple Data Types

```
int
unsigned int

char


float
double
```

- Integer data
  - Unsigned version can store only non-negative values
- Character data
- Floating point data

```
const
```

- Constant modifier
  - Can be used to prefix simple data types
    - E.g. `const int  i = 123;`
  - Value must be initialized during declaration and cannot be changed afterwards

Simple Data Types [new]

```
bool
```

- Boolean data
  - Can have the value **`true`** or **`false`** only
  - Internally, **`true`** = 1, **`false`** = 0
  - Can be used in a condition
  - Improve readability
  - Reduce error

```
bool done = false;

while (!done) {        "While not done"

    if (...)
        done = true;   "Condition met, I'm done"
}
```

Example Usage

Array

- A collection of **homogeneous** data
  - Data of the same type

```
int iA[10];
```

```
iA[0] = 123;               Store value into 1st element

iA[9] = 456;               Store value into last, 10th element

iA[1] = iA[0] + iA[9];     Store and read values
```

Example Usage

Array

- **Limitation:**
  - A function return type cannot be an array
  - An array parameter is “passed by address”
  - An array cannot be the target of an assignment

```
int[10] someFunction( ) {...}

int ia[10], ib[10];

ia = ib;
```

Error: **cannot return array**

Error: **array assignment is invalid**

Structure

- A collection of **heterogeneous** data
  - Data of different type
  - Should be a collection describing a common entity

```
struct Person {
    char name[50];
    int age;
    char gender;
};

Person s1;
```

- Declaration: A structure to store information about a person:
  - **Name**: String of 50 characters
  - **Age**: integer
  - **Gender**: 'm' = male; 'f' = female
- **s1** is a structure variable
- Additional Note:
  - In C, you need to write:

```
struct Person s1;
```

Structure

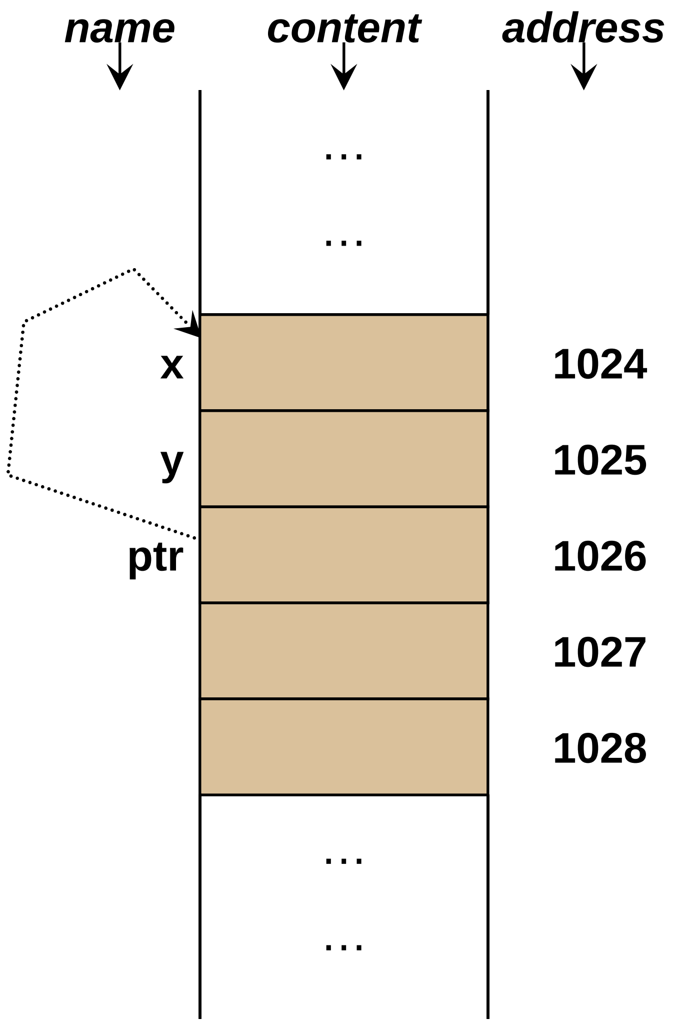
```
Person s1 = { "Potter", 13, 'm' };    Declare & Initialize
Person s2;                             Declare only

s2 = s1;    Structure assignment. Everything copied.

s1.age = 14;    Use '.' to access a field

s2.age = s1.age * 2;    Store and read a field

s2.gender = 'f';
```

Example Usage

Pointer

- A pointer variable contains the address of a memory location

```
int x;

int *ptr;

ptr = &x;

*ptr = 123;
```



- Note the different meanings of *
  1. Declaring a pointer
  2. Dereference a pointer

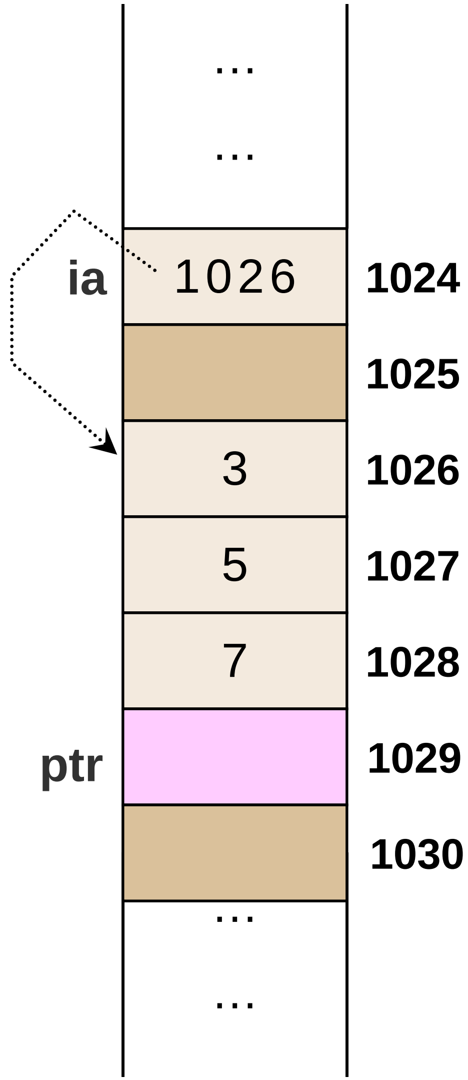
Pointers and Arrays

- Array name is a **constant pointer**
  - Points to the zeroth element

```
int ia[3] = {3, 5, 7};
```

- Is the following valid?

```
int* ptr;

ptr = ia;
ia = ptr;
ptr[2] = 9;

ptr = &ia[1];
ptr[1] = 11;
```

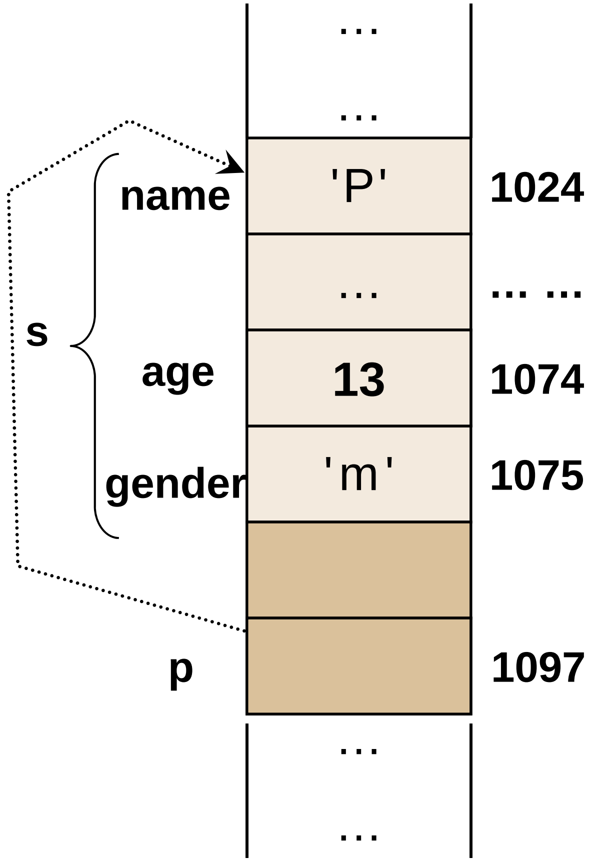


Pointer and Structure

- Pointer can points to a structure as well

```
int main()
{
   Person s =
           { "Potter", 13, 'm' };

   Person *p; //Person Pointer


   p = &s;


   p->age = 14;

   (*p).age = 14;

}
```

Equivalent Statements



Dynamic Memory Allocation : **new**

- **New** memory box can be allocated at **runtime**
  - Using the **`new`** keyword

SYNTAX

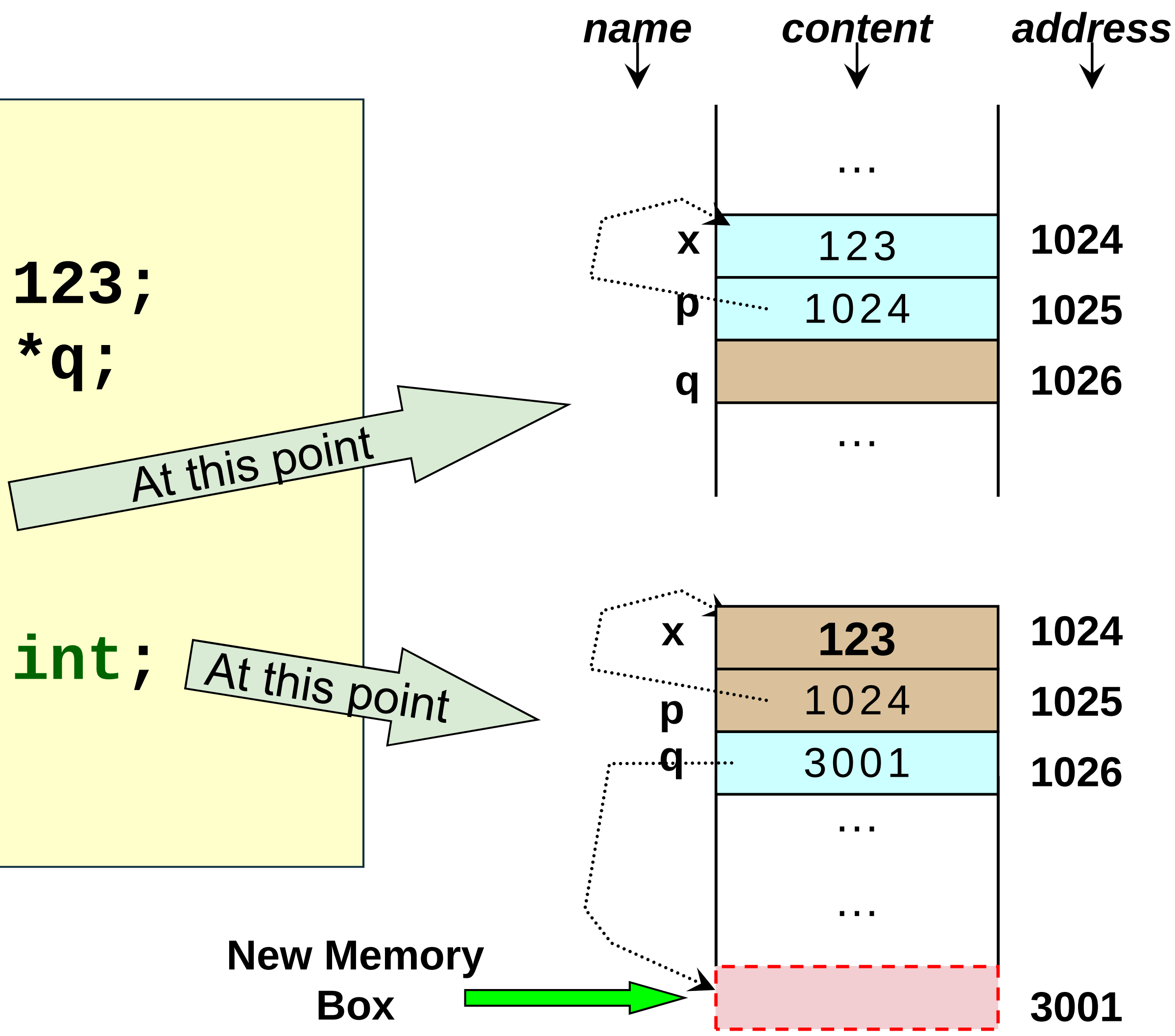
```
new data_type;
```

- ***`data_type`*** can be
  - Predefined datatype: **`int`**, **`float`**, **`array`**, etc
  - User defined datatype: structure or class

- **Address** of the newly allocated memory boxes are then returned
  - Usually, a pointer variables is used to store the address

new : Single Element

```
int main()
{
    int x = 123;
    int *p, *q;

    p = &x;

    q = new int;

}
```

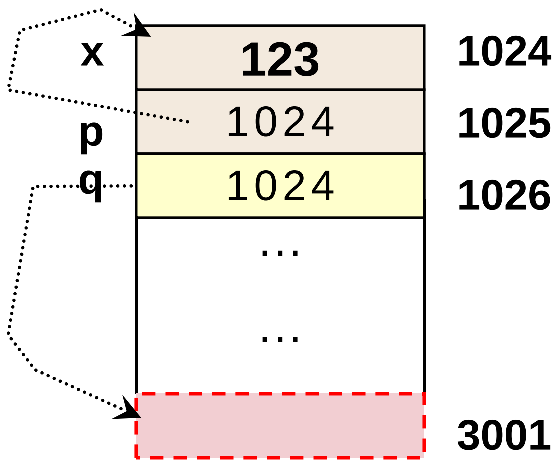


new : Single Element

- **Important:**
  - q is the **only** variable storing the address of the new memory boxes
  - If q is changed, the new location is **lost** to your program, known as **memory leak**

```
int main()
{
   int x = 123;
   int *p, *q;
   p = &x;

   q = new int;
   q = p;

}
```

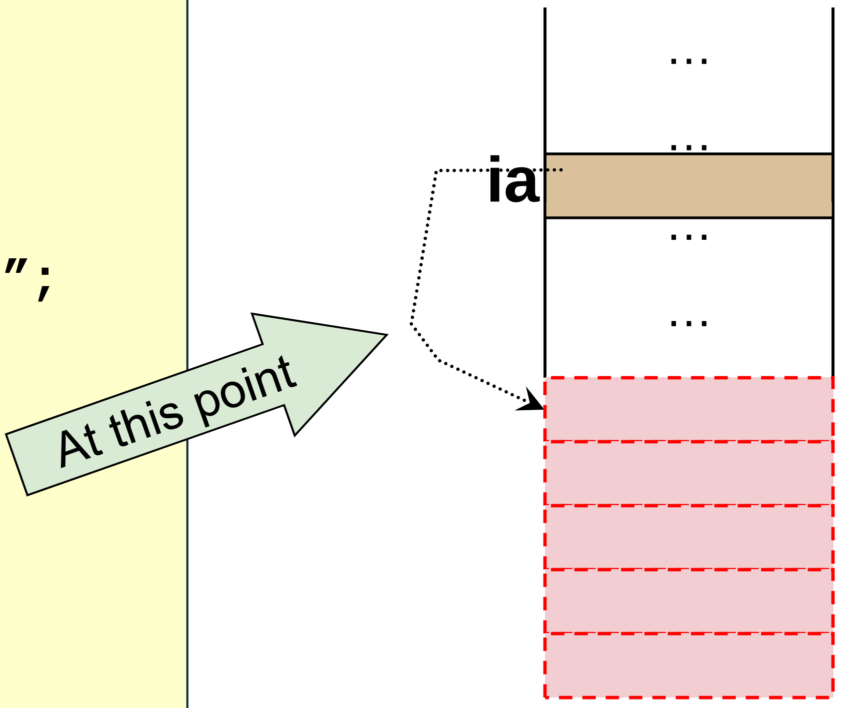
At this point



new : Array of elements

- Whole array can be allocated dynamically
  - The size can be supplied at run time

```
int main()
{
   int size;
   int *ia;

   cout << "Enter size:";
   cin >> size;

   ia = new int[size];

   ia[0] = ...
   ia[1] = ...

}
```



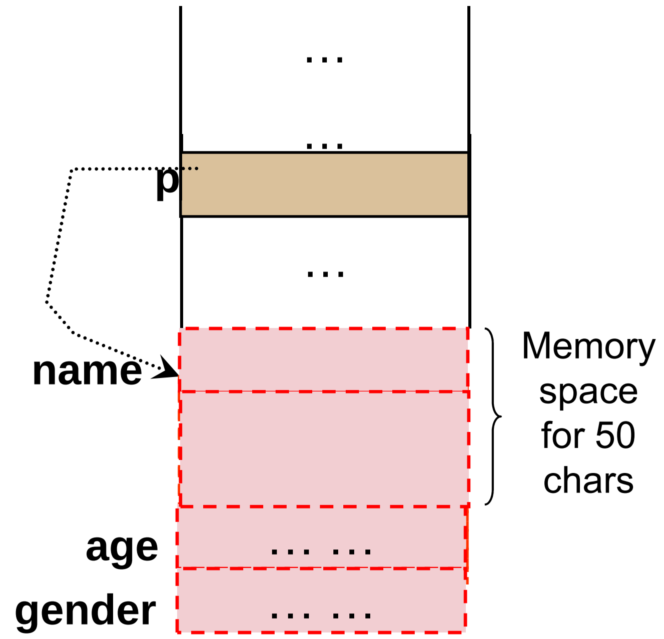
Assume size = 5

new : Structure

- Dynamic allocation for structure or object are both possible

```
int main()
{
    Person *p;

    p = new Person;


    p->age = 14;

    (*p).age = 14;

}
```

At this point



Releasing memory to system : `delete`

- Dynamically allocated memory can be returned to the system (unallocated)
  - Using `delete` keyword

SYNTAX

```
delete pointer
delete [ ] pointer_to_array
```

- Memory box(es) pointed by the pointer will be returned to the system

- **Important:**
  - Dereferencing pointer after **`delete`** is invalid!
  - Make sure you use **`delete []`** for deleting an array

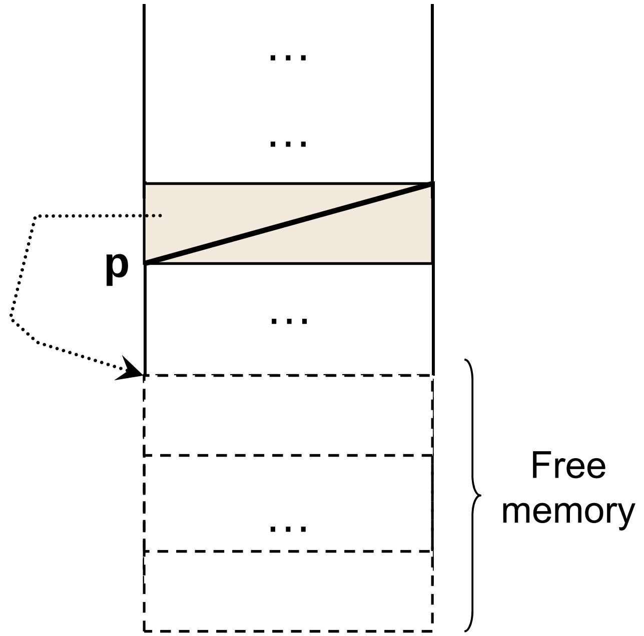
delete : An example

```
int main()
{
    Person *p;

    p = new Person;


    p->age = 14;


    delete p;
    p = NULL;


    p->age = 14;

}
```

At this point

Good Practice: **Always** set a pointer to NULL after delete

Error!



General Advices on using Pointers

- Incorrect / Careless use of pointers can make your life ***miserable***:
  - Program Crashes (Runtime Error):
    - Segmentation Fault / Bus Error
  - "Weird" behavior:
    - Program works erratically ☹
- Useful Guidelines:
  - **Always** initialize a pointer
    - Set to **NULL**
    - When:
      - Declaring a new pointer
      - After memory deallocation
  - **Make sure the pointer is pointing to a right place!**
    - Take care when deleting:
      - Anyone else pointing to the same place?

Function

Modular Programming

Function

- Organize useful programming logic into a unit
  - **Self contained:**
    - only relies on parameter for input
    - output is well defined
  - **Portable**
  - **Ease of maintenance**

```
int factorial( int n )
{
    int result = 1, i;
    for ( i = 2; i <= n; i++ )
      result *= i;

    return result;
}
```

Function Prototype and Implementation

- Good practice to provide function prototypes

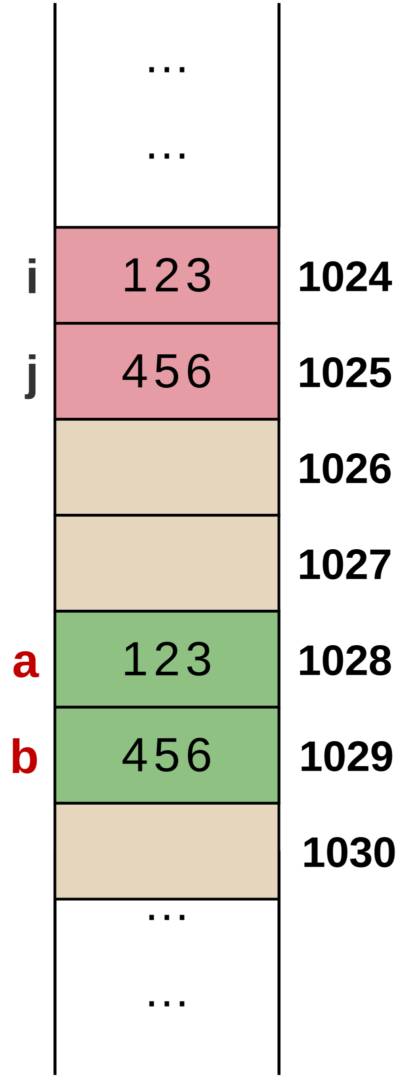
```
int factorial( int );

int main( )
{
    ...
}

int factorial( int n )
{
    int result = 1, i;
    for (i = 2; i <= n; i++)
      result *= i;

    return result;
}
```

Function : Parameter Passing

- There are **three** ways of passing a parameter into a function:
  1. **Pass by value**
  2. **Pass by address** or **Pass by pointer**
     - Known as “Pass by reference” in CS1101C, which is technically incorrect ☺
  3. **Pass by reference** [new]
- Lets try to define a function **`swap(a, b)`** to swap the parameters
  - Desired behavior: value of **`a`** and **`b`** swapped after function call

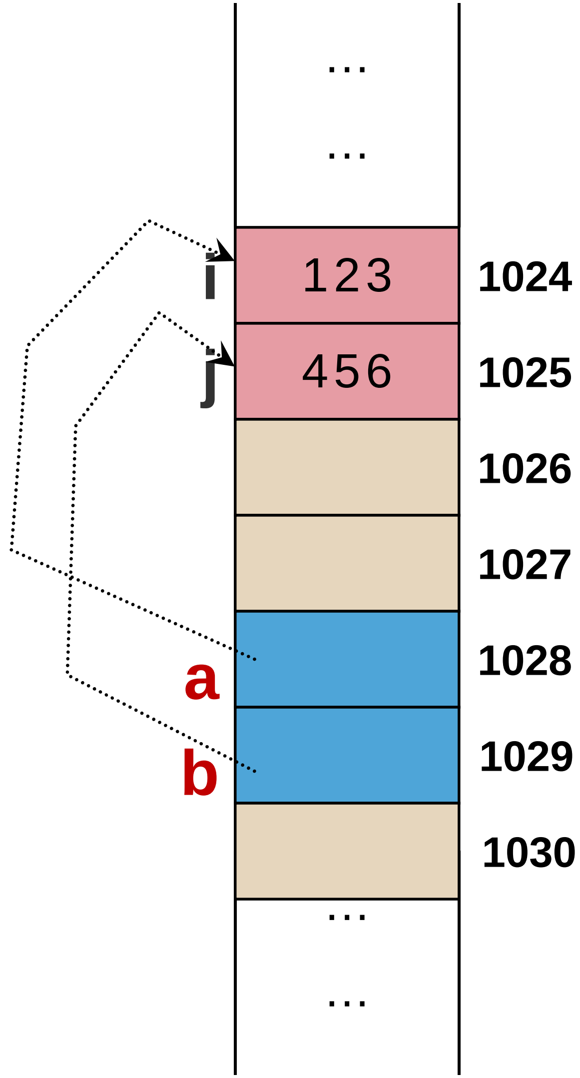
Function : Pass by value

```
void swap_ByValue( int a, int b )
{   int temp;

    temp = a;
    a = b;
    b = temp;
}

int main()
{
    int i = 123, j = 456;

    swap_ByValue( i, j );

    cout << i << endl;
    cout << j << endl;
}
```



Function : Pass by address/pointer

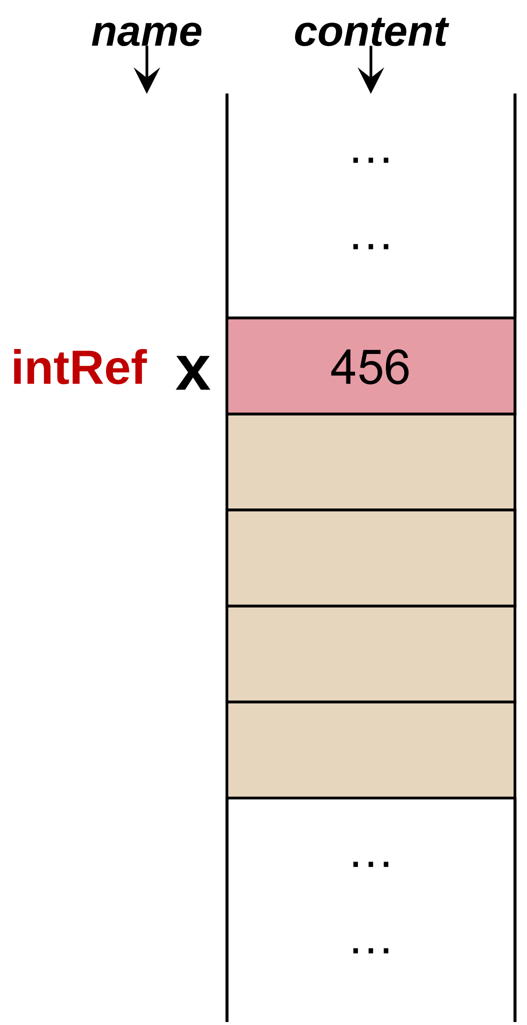
```
void swap_ByAdr( int* a, int* b )
{   int temp;

    temp = *a;
    *a = *b;
    *b = temp;
}

int main()
{
    int i = 123, j = 456;

    swap_ByAdr( &i, &j );

    cout << i << endl;
    cout << j << endl;
}
```



Reference [new]

- A reference is an ***alias*** (alternative name) for a variable

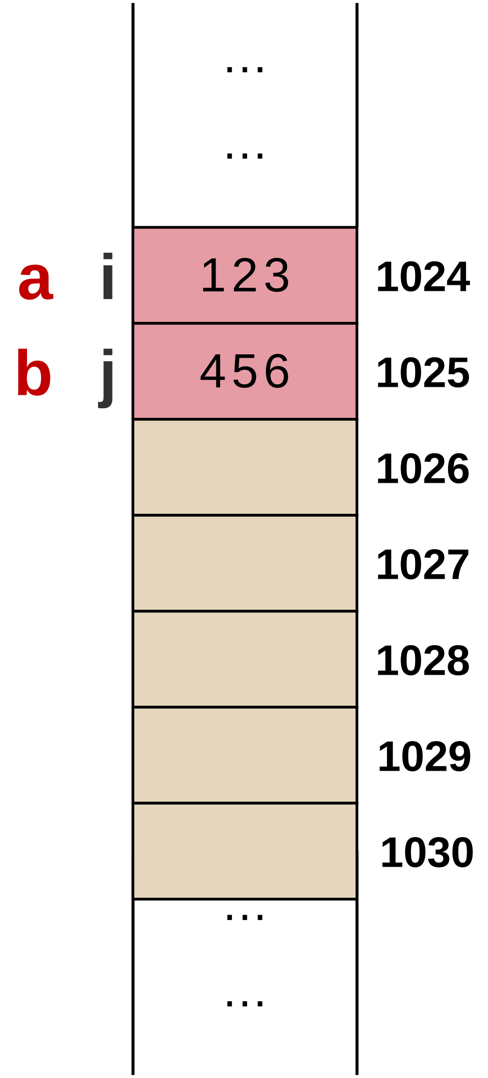
```
int x = 456;

int& intRef = x;

intRef++;
cout << x << endl;     //result?
```




```
int& intRef;

int I;
int& ref = &I;
```

Function : Pass by reference [new]

```
void swap_ByRef( int& a, int& b )
{   int temp;

    temp = a;
    a = b;
    b = temp;
}

int main()
{
    int i = 123, j = 456;

    swap_ByRef( i, j );

    cout << i << endl;
    cout << j << endl;
}
```



Function : Passing Parameters

- **By Value:**
  - Simple data types (`int`, `float`, `char` etc) and structures are passed by value
  - Cannot change the actual parameter
- **By Address:**
  - Requires the caller to pass in the address of variables using "**&**"
  - Requires dereferencing of parameters in the function
  - Arrays are pass by address
- **By Reference:**
  - No additional syntax except to declare the parameters as references
  - No additional memory storage
    - Faster execution and less memory usage

Useful Library

Can’t live without them

C Libraries in C++

- Most C standard libraries are ported over in C++
  - Minor change in library name
    - **`<math.h>`** is now **`<cmath>`**
    - **`<stdlib.h>`** is now **`<cstdlib>`**
    - Etc

- No need for **`–lm`** when using **`cmath`** library

Summary

- Control Statement
- Declaration
  - Simple Data Type
  - Composite Data Type
  - Pointers
- Function
- Useful C Libraries in C++

Reading Materials

- Carrano’ s Book
  - Appendix A: pages 813 – 888
  - Review of C++ Fundamentals

For Your Own Reading

Potentially useful topics

Enumeration [new]

- Enumeration allows the programmer to declare a **new data type** which take **specific values only**

```
enum Color {
    Red, Yellow, Green
};
```

Color is a new data type

Values that are valid for a **Color** variable

Example Declaration

```
Color c1, c2;

c1 = Yellow;
c2 = c1;

c1 = 123;

c2++;
```

Error: **c1** is not an integer

Error: **++** is not defined for enumeration

Example Usage

Enumeration [new]

```
Color myColor;

...

switch (myColor) {
    case Red:
        ...
    case Yellow:
        ...
    case Green:
        ...
}

int myInt;
myInt = myColor;


Color newColor;
newColor = Color(1);
```

enum can be used in a switch statement

enum can be converted to integer
By default, 1st value == 0, 2nd value == 1 etc.
i.e. Red = 0, Yellow = 1, …

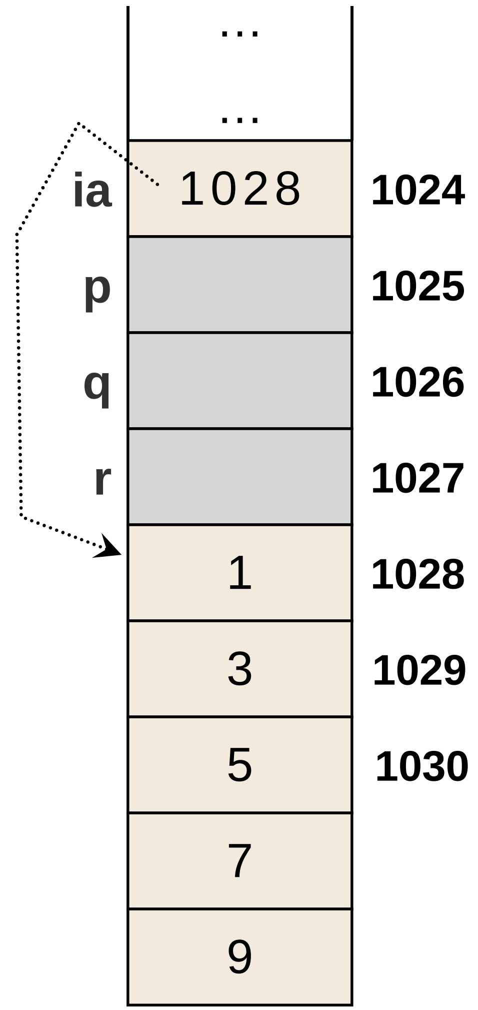
Similarly, integer can be converted to **enum** type

newColor will have the value **Yellow** in this case

Pointer Arithmetic [expanded]

- Addition and subtraction of pointers are **valid**

```
int ia[5] = {1, 3, 5, 7, 9};
int *p = ia;
int *q, *r;

q = p + 3;     //what is q?
r = q – 1;     //what is r?

cout << *p << endl;
cout << *q << endl;
cout << *r << endl;

cout << *p + 1 << endl;
cout << *(p + 1) << endl;
```



Pointer Arithmetic [expanded]

- Two forms of element access for arrays:

```
int ia[5] = {1, 2, 3, 4, 5};

for (int i = 0; i < 5; i++)
      cout << ia[i] << endl;
```

Using indexing

```
int ia[5] = {1, 2, 3, 4, 5};
int *ptr;

for (ptr = ia; ptr < ia + 5; ptr++)
      cout << *ptr << endl;
```

Using pointer arithmetic

Function : Default Argument [new]

- In C++, function parameter can be given a default value
  - Default is used if the caller does not supply actual parameter

```
double logarithm( double N, double base = 10 )
{  ... Calculates Logbase(N) ...              }

int main( )
{
   cout << logarithm(1024,2) << endl;
   cout << logarithm(1024)   << endl;
}
```

Function Overloading [new]

- Compiler recognizes function by the **function signature**
  - Function name + data types of parameters
- Example:
  - `factorial( int )`
  - `sqrt( double )`
- In C++, multiple versions for a function is allowed
  - Function name is the same
  - Parameter number and/or type must be different, i.e. different function signature
  - Known as **function overloading**

Function Overloading [new]

```
int maximum( int a, int b )
{
    if (a > b)  return a;
    else        return b;
}

int maximum( int a, int b, int c )
{
      return maximum( maximum(a, b), c);
}

double maximum( double a, double b )
{
    if (a – b > 0.00001) return a;
    else                 return b;
}
```