MACC
Matemáticas Aplicadas y
Ciencias de la Computación

Universidad del Rosario

Introduction to C++: dynamic memory management

HPC School — June 10, 2019

MA:CC:UR:OS:AR:IO

Julián J. Rincón (@julianjrincon)
Department of Applied Mathematics and Computer Science
Universidad del Rosario, Bogotá - Colombia

Goal

- Be able to understand code like the following...

```
1  const size_t dim = 101;
2  int *ptr = new int[dim];
3
4  // preconditions
5
6  for (size_t i = 0; i < dim; i++) {
7      ptr[i] = i * i;
8      cout << ptr[i] << endl;
9  }
10
11 // postconditions
12
13 delete[] ptr;
14 ptr = nullptr;
```

Outline

1) Variables and operators

2) Expressions and statements

3) Functions

4) Pointers and arrays

C++

- The simplest code we can type in C++

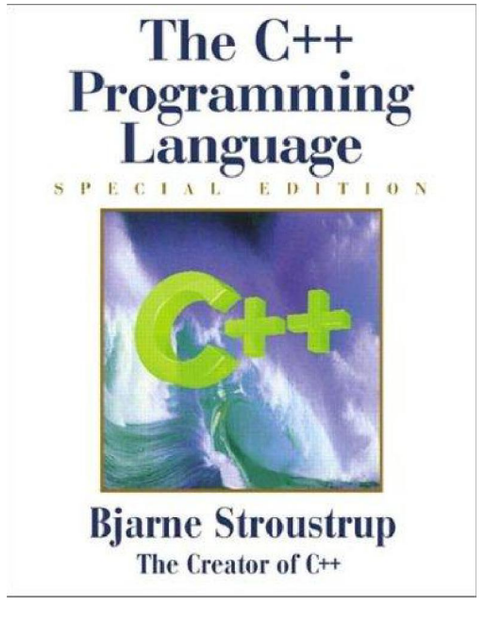
```
1 int main() {
2     return 0;
3 }
```

C++

- The simplest code we can type in C++

```
1  int main() {
2      return 0;
3  }
```

- Facts about C++:
  - ✓ developed by Bjarne Stroustrup in 1983
  - ✓ main purpose was systems programming
  - ✓ has a gigantic standard library
  - ✓ last standard was C++17



C++

- The simplest non-trivial code we can type in C++

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     cout << "Hello, World!" << endl;
6     return 0;
7 }
```

C++

- The simplest non-trivial code we can type in C++

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

int main() {
    cout << "Hello, World!" << endl;
    return 0;
}
```

- Compilation

```
g++ hello_world.cpp
g++ hello_world.cpp -o hello_world
```

```
/*
 * File: PowersOfTwo.cpp
 * ---------------------
 * This program generates a list of the powers of
 * two up to an exponent limit entered by the user.
 */

#include <iostream>
using namespace std;

/* Function prototypes */

int raiseToPower(int n, int k);

/* Main program */

int main() {
   int limit;
   cout << "This program lists powers of two." << endl;
   cout << "Enter exponent limit: ";
   cin >> limit;
   for (int i = 0; i <= limit; i++) {
      cout << "2 to the " << i << " = "
           << raiseToPower(2, i) << endl;
   }
   return 0;
}

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

Program comments

Library inclusions

Function prototype

Main program

Function definition

Variables

- Definition, and definition and initialization

```
1 <type-specifier> name;
2 <type-specifier> name = value;
```

- Examples

```
1 int x;
2 char cc = 'Z';
3 const double PI = 3.14159265358979323846;
```

Conventions for Variable Names

- A variable's name should give some indication of its meaning
- Variable names are normally lowercase
- Multiple words should visually distinguish each word
- Variable naming is case-sensitive

Built-in types

Type	Meaning	Minimum Size
`bool`	boolean	NA
`char`	character	8 bits
`wchar_t`	wide character	16 bits
`char16_t`	Unicode character	16 bits
`char32_t`	Unicode character	32 bits
`short`	short integer	16 bits
`int`	integer	16 bits
`long`	long integer	32 bits
`long long`	long integer	64 bits
`float`	single-precision floating-point	6 significant digits
`double`	double-precision floating-point	10 significant digits
`long double`	extended-precision floating-point	10 significant digits

Keywords

```
alignas     continue      friend     register           true
alignof     decltype      goto       reinterpret_cast   try
asm         default       if         return             typedef
auto        delete        inline     short              typeid
bool        do            int        signed             typename
break       double        long       sizeof             union
case        dynamic_cast  mutable    static             unsigned
catch       else          namespace  static_assert       using
char        enum          new        static_cast        virtual
char16_t    explicit      noexcept   struct             void
char32_t    export        nullptr    switch             volatile
class       extern        operator   template           wchar_t
const       false         private    this               while
constexpr   float         protected  thread_local
const_cast  for           public     throw
```

```
and         bitand      compl     not_eq     or_eq      xor_eq
and_eq      bitor       not       or         xor
```

Some operators...

- Arithmetic

Operator	Function	Use
`+`	unary plus	`+ expr`
`-`	unary minus	`- expr`
`*`	multiplication	`expr * expr`
`/`	division	`expr / expr`
`%`	remainder	`expr % expr`
`+`	addition	`expr + expr`
`-`	subtraction	`expr - expr`

- Relational

Associativity	Operator	Function	Use
Right	`!`	logical NOT	`!expr`
Left	`<`	less than	`expr < expr`
Left	`<=`	less than or equal	`expr <= expr`
Left	`>`	greater than	`expr > expr`
Left	`>=`	greater than or equal	`expr >= expr`
Left	`==`	equality	`expr == expr`
Left	`!=`	inequality	`expr != expr`
Left	`&&`	logical AND	`expr && expr`
Left	`||`	logical OR	`expr || expr`

Assignment operator

- Binary, right-associative operator
- Not an equivalence relation
- Left operand is typically a variable
- Right operand can be a literal, a variable, or an expression

Assignment operator

- Binary, right-associative operator
- Not an equivalence relation
- Left operand is typically a variable
- Right operand can be a literal, a variable, or an expression

```
1 int t;
2 double rc, sc;
3
4 rc = 4 * 4.4 + 5 / sqrt(z);
5 t = 33;
6 sc = rc;
```

Expressions

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4
5  int main(void){
6      float a = 1.4, b = 3.1, c = 0.9;
7      // expression: thing between = and ;
8      float sol = (-b + sqrt(b * b - 4 * a * c))
9                  / (2 * a);
10     // print to standard output
11     cout << "Solution " << sol << endl;
12     return 0;
13 }
```

Operator precedence

Operators organized into precedence groups	Associativity
`()` `[]` `->` `.`	*left*
unary operators: `-` `++` `--` `!` `&` `*` `~` `(`*type*`)` `sizeof`	*right*
`*` `/` `%`	*left*
`+` `-`	*left*
`<<` `>>`	*left*
`<` `<=` `>` `>=`	*left*
`==` `!=`	*left*
`&`	*left*
`^`	*left*
`|`	*left*
`&&`	*left*
`||`	*left*
`?:`	*right*
`=` *op*`=`	*right*

Basic statements

- Basic statements

```
1  // expression statement
2  long ave = (min + max) / 2;
3
4  // null statement
5  ;
6
7  // block statement
8  {
9      sum += val
10     val++;
11 }
```

Conditional statements

- Conditional execution based on a decision
- The if/else statement

```
1 if (condition_1){
2       code_1;
3 } else if(condition_2) {
4       code_2;
5 } else {
6       code_3;
7 }
```

Iterative statements

- Also called loops, repetitive execution according to condition
- The while and do/while statements

```
1 while (condition) {
2     code;
3 }
```

```
1 do {
2     code;
3 } while (condition);
```

- The for statement

```
1 for (init-stmt; condition; expr) {
2     code;
3 }
```

Break and continue statements

- A `break` statement terminates the nearest enclosing `while`, `do/while`, or `for` statement
  - can appear only within an iteration statement
- A `continue` statement terminates the current iteration of the nearest enclosing loop and immediately begins the next iteration
  - can appear only inside a `for`, `while`, or `do/while` loop, including blocks nested inside such loops
- Execution resumes at the statement immediately following the terminated statement

Functions

```
1 <return-type> name(<params>) {
2     code;
3     return <expr>;
4 }
```

- Example

```
1 int fact(int val) {
2     int ret = 1;
3     while (val > 1) {
4         ret *= val--;
5     }
6     return ret;
7 }
```

Functions

```
1 <return-type> name(<params>) {
2     code;
3     return <expr>;
4 }
```

- Example

```
1 int main() {
2     int j = fact(5);
3     cout << "5! is " << j << endl;
4     return 0;
5 }
```

Input/output

- To write to the screen (standard output)

```
1 #include <iostream>
2 using namespace std;
3
4 // ...
5
6 cout << "This␣is␣a␣message␣in␣C++␣" << endl;
```

Input/output

- To read from the keyboard (standard input)

```
1 #include <iostream>
2 using namespace std;
3
4 // ...
5
6 int r;
7 cout << "Enter␣an␣integer␣:";
8 cin >> r;
9 cout << "The␣number␣is␣" << r << endl;
```

Input/output

- To write to a text file

```
1  #include <fstream>
2  using namespace std;
3
4  int main() {
5      ofstream myfile;
6      myfile.open("example.txt");
7      myfile << "This␣is␣a␣line.\n";
8      myfile << "Another␣line.␣";
9      myfile << "More␣lines?.␣";
10     myfile.close();
11     return 0;
12 }
```

Input/output

- To read from text file

```
1  #include <iostream>
2  #include <fstream>
3  #include <string>
4  using namespace std;
5
6  int main () {
7      string line;
8      ifstream myfile("example.txt");
9      while (getline(myfile, line))
10         cout << line << '\n';
11     myfile.close();
12     return 0;
13 }
```

C++ memory Model

- How information is stored in a computer?
  - Information is stored as combinations of *bit*s
    - 1 bit [b] = contraction("binary digit")
      - takes two values: 0/1; off/on; false/true
    - 1 byte [B] = 8 bits — 1B = 8b [= `sizeof(char)`]
    - 1 word [w] = `sizeof(int)` [= 4B or 8B], defined like that on most machines

Binary and hexadecimal representations

- Bits in an unsigned integer are encoded using *binary notation*, that is, 0's and 1's
- Booleans seem to be naturally represented by one single bit. Is this the case in modern computers?
- Hexadecimal notation is mostly used to refer to memory addresses in the RAM

C++ memory layout

- Memory is usually specified in terms of bytes

- Memory is divided into three main regions.
  Can you name them?

Roberts, Eric. (2013). *Programming Abstractions in C++*. Pearson.

C++ memory layout

- Memory is usually specified in terms of bytes
- Memory is divided into three main regions.
  Can you name them?

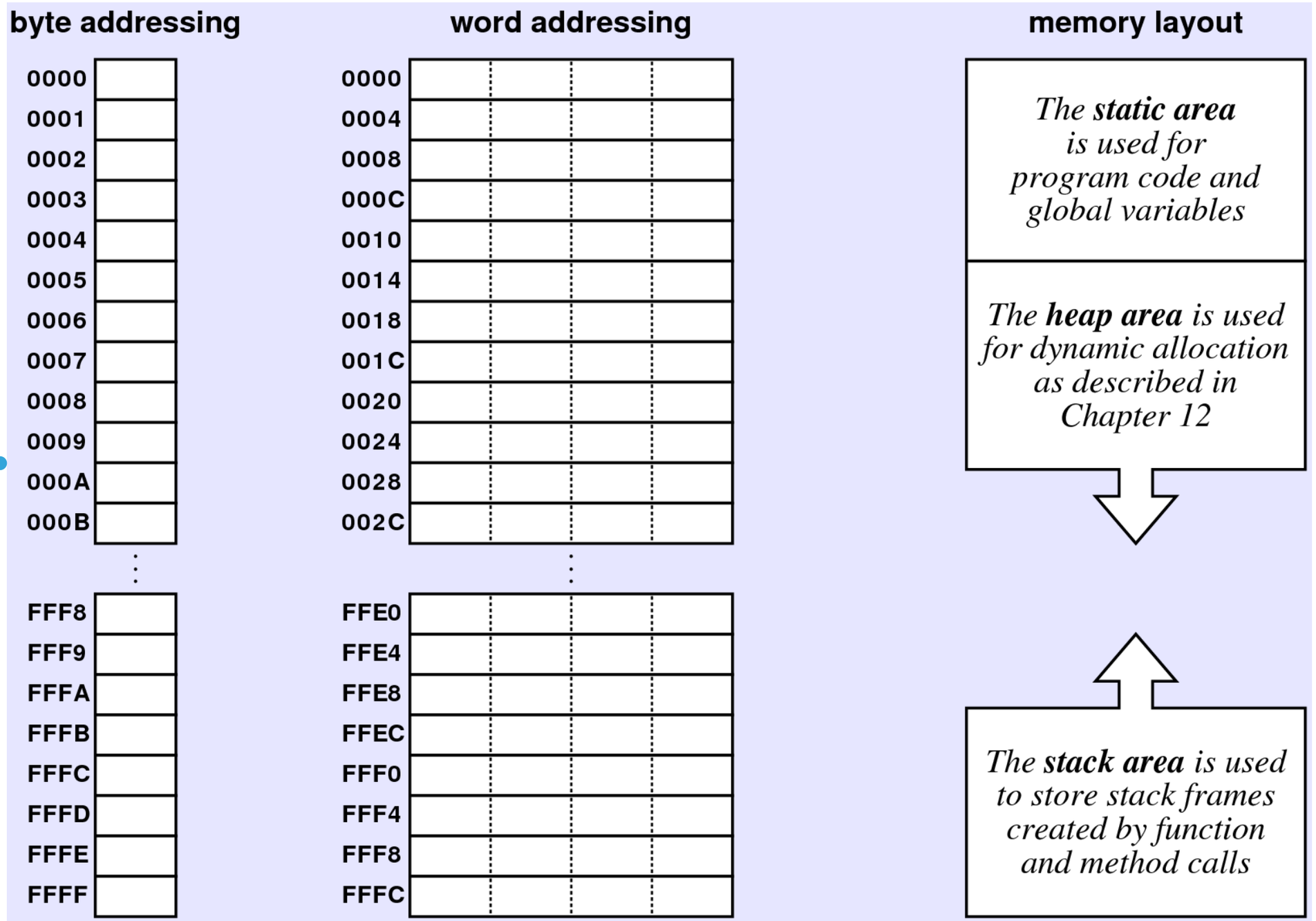
- Static area: Code and global and static variables
- Heap area: Dynamic allocation
- Stack area: Stack frames, local variables

Roberts, Eric. (2013). *Programming Abstractions in C++*. Pearson.

C++ memory layout

- Memory is usually specified in terms of bytes

Memory diagrams



Assigning memory to variables

- Where is this variable stored?

```
const double PI = 3.15159;
```

- What about this one?

```
map<string, bool>::const_iterator it = mp.cbegin();
```

- And this other one?

```
Complex pt = new Complex(3, 4);
```

Example: powers of two

- Let's analyze, step by step, the following code

```
int main() {
    int limit;
    cout << "Enter exponent limit: ";
    cin >> limit;
    for (int i = 0; i <= limit; i++)
        cout << "2 ^ " << i << " = " << toPower(2, i) << endl;

    return 0;
}

int toPower(int n, int k) {
    int result = 1;
    for (int i = 0; i < k; i++) result *= n;

    return result;
}
```

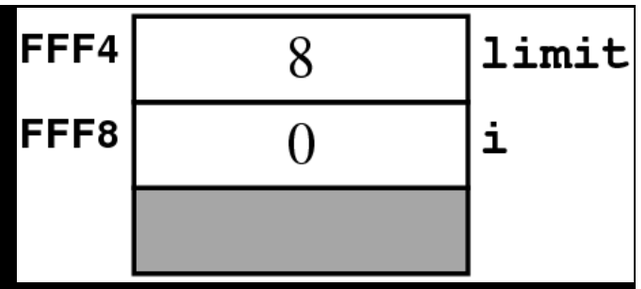
Example: powers of two — main

- Start with main function, of course

```
int main() {
    int limit;
    cout << "Enter exponent limit: ";
    cin >> limit;
    for (int i = 0; i <= limit; i++)
        cout << "2 ^ " << i << " = " << toPower(2, i) << endl;

    return 0;
}

int toPower(int n, int k) {
    int result = 1;
    for (int i = 0; i < k; i++) result *= n;

    return result;
}
```

Example: powers of two — main

```
int main() {
    int limit;
    cout << "Enter exponent limit: ";
    cin >> limit;
    for (int i = 0; i <= limit; i++)
        cout << "2 ^ " << i << " = " << toPower(2, i) << endl;

    return 0;
}

int toPower(int n, int k) {
    int result = 1;
    for (int i = 0; i < k; i++) result *= n;

    return result;
}
```



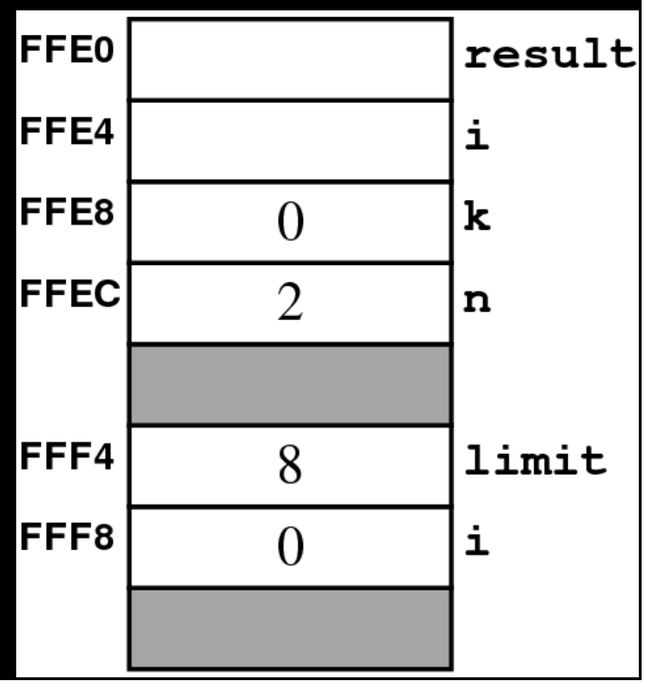
Example: powers of two — two power

- Now the auxiliary function

```
int main() {
    int limit;
    cout << "Enter exponent limit: ";
    cin >> limit;
    for (int i = 0; i <= limit; i++)
        cout << "2 ^ " << i << " = " << toPower(2, i) << endl;

    return 0;
}

int toPower(int n, int k) {
    int result = 1;
    for (int i = 0; i < k; i++) result *= n;

    return result;
}
```

Example: powers of two — two power

```
int main() {
    int limit;
    cout << "Enter exponent limit: ";
    cin >> limit;
    for (int i = 0; i <= limit; i++)
        cout << "2 ^ " << i << " = " << toPower(2, i) << endl;

    return 0;
}

int toPower(int n, int k) {
    int result = 1;
    for (int i = 0; i < k; i++) result *= n;

    return result;
}
```



Pointers

- hold the address in memory of a variable
- are data items whose value is an address in memory
- have the same data type as the variable they have address
- allow you to refer to a large data structure in a compact way
- make it possible to manage and reserve memory during program execution
- can be used to record relationships among data items

Declaring pointers variables

- To declare a variable as a pointer use the asterisk '*'

```
int * pointer;
char * cptr;
```

- declares `pointer` and `cptr` to be types pointer-to-`int` and pointer-to-`char`
- Syntactically, the '*' belongs to the variable not the type
- What is the difference between these two statements?

```
double *ptr1, *ptr2;
double *ptr1, ptr2;
```

Declaring pointers variables

- To declare a variable as a pointer use the asterisk '*'

```
int * pointer;
char * cptr;
```

- declares `pointer` and `cptr` to be types pointer-to-`int` and *are distinct types, though they are represented as addresses*
- Syntactically, the '*' belongs to the variable not the type
- What is the difference between these two statements?

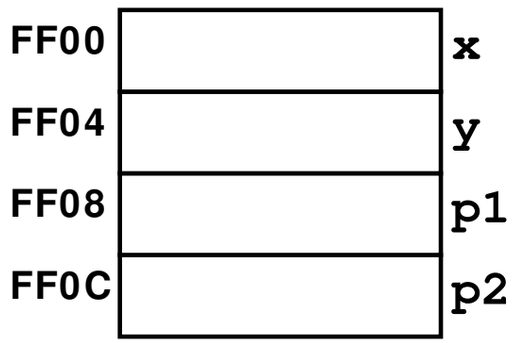
```
double *ptr1, *ptr2;
double *ptr1, ptr2;
```

Address-of and dereferencing operatorS

- '&': address-of
  - returns the memory address in which the value was stored
- '*': value-pointed-to
  - returns the value to which the pointer points to
- To understand these operators, consider the next example

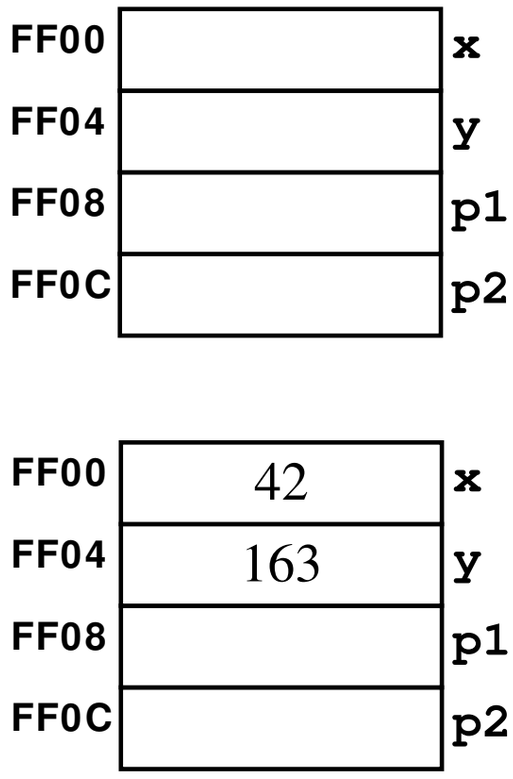
Example: pointers

```
int x, y;
int *p1, *p2;
```



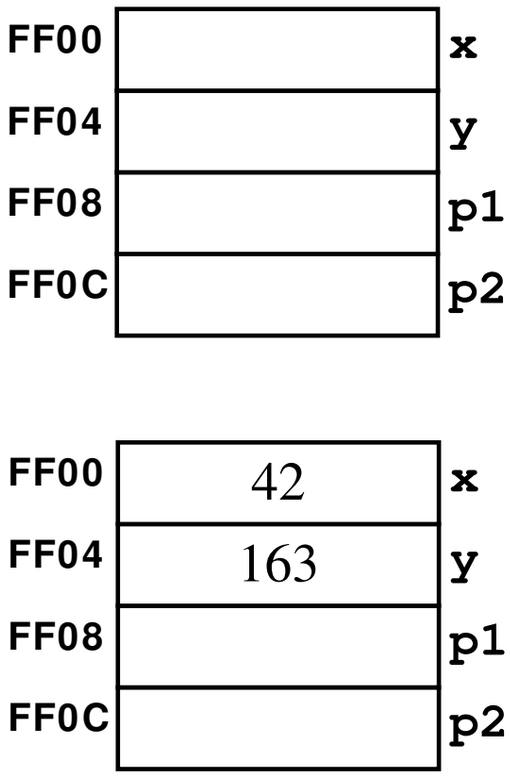
Example: pointers

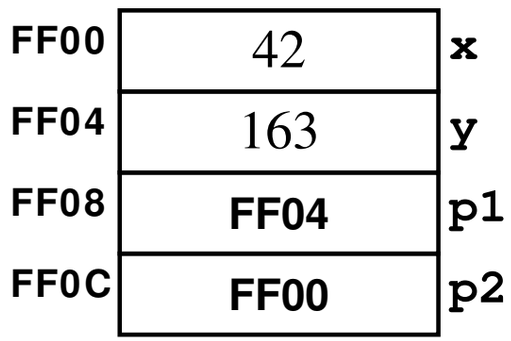
```
int x, y;
int *p1, *p2;

x = 42;
y = 163;
```

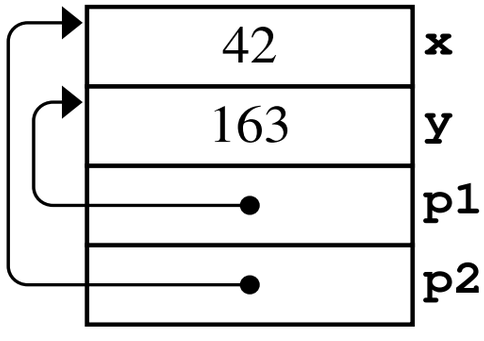


Example: pointers

```
int x, y;
int *p1, *p2;

x = 42;
y = 163;

p1 = &y;
p2 = &x;
```

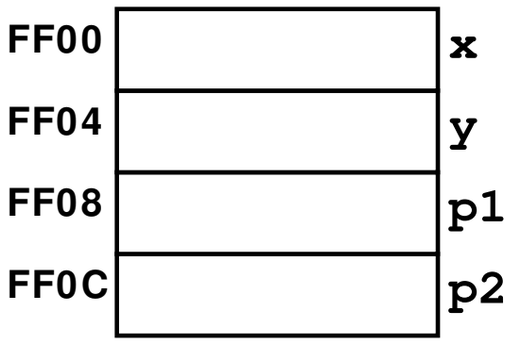


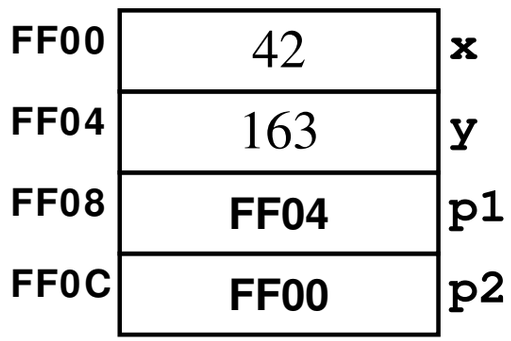


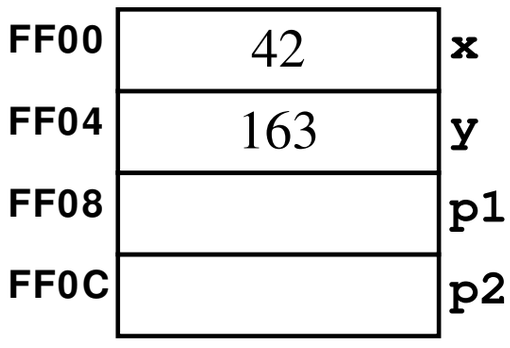


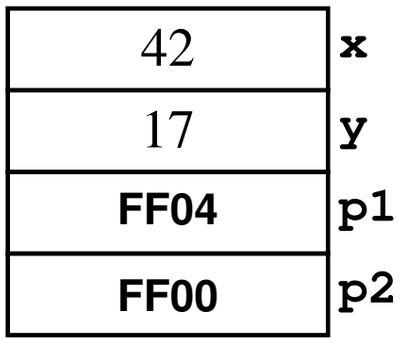
Example: pointers

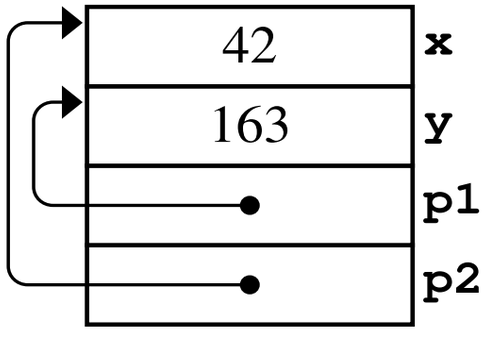
```
int x, y;
int *p1, *p2;

x = 42;
y = 163;

p1 = &y;
p2 = &x;

*p1 = 17;
```





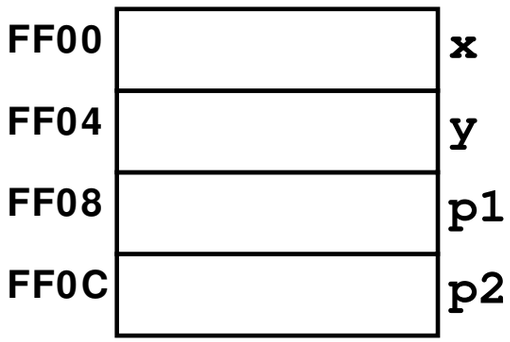


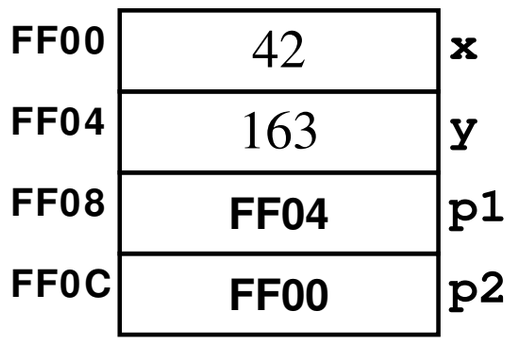


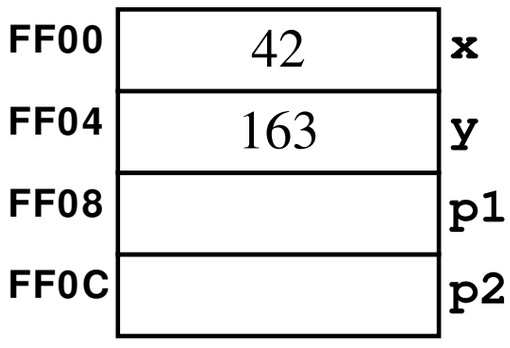


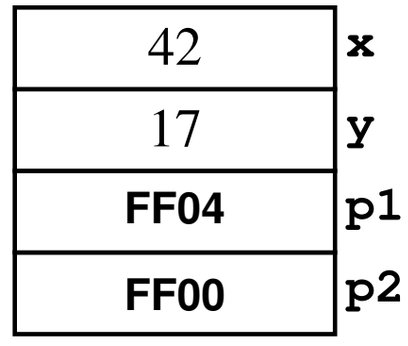
Example: pointers

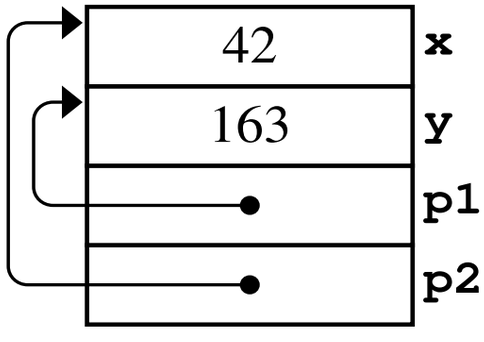
```
int x, y;
int *p1, *p2;

x = 42;
y = 163;

p1 = &y;
p2 = &x;

*p1 = 17;

p1 = p2;
```

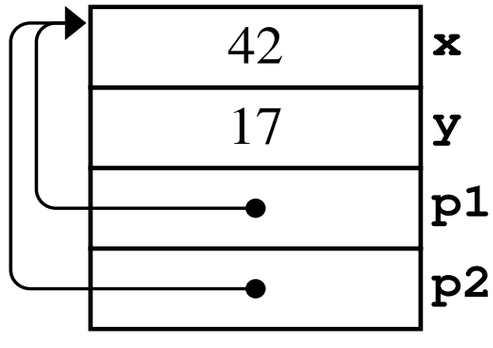












The NULL pointer [Old!]

- Pointer value that indicates that the pointer does not in fact refer to any valid memory address
- It is illegal to use the dereferencing operator (*) on NULL
- Keyword: NULL
- Defined in the interface <cstddef>

```
Point * pointer_to_pt;
// some action done on pointer_to_pt
if (pointer_to_pt == NULL)
     cerr << “Something's NOT right. Check!"
```

The NULLptr pointer

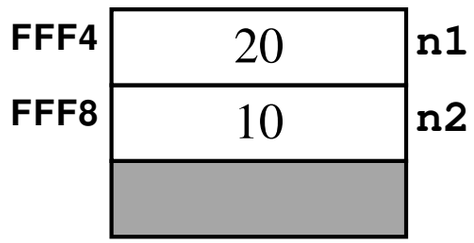
- A null pointer does not point to any object
- Code can check whether a pointer is null before attempting to use it
- **nullptr** is a literal that has a special type that can be converted to any other pointer type

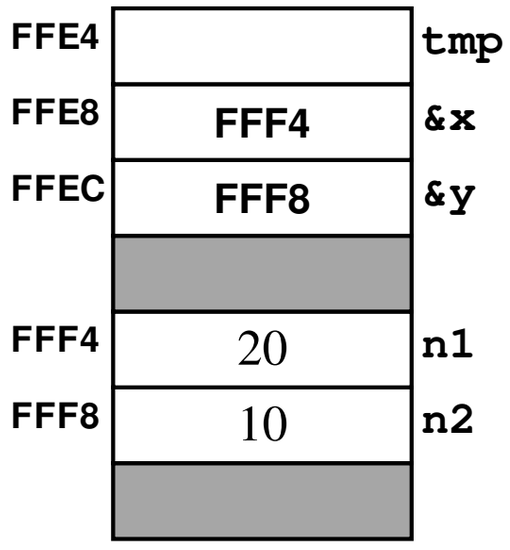
```
int *p1 = nullptr; // equivalent to int *p1 = 0;
// must #include <cstdlib>
int *p3 = NULL; // equivalent to int *p3 = 0;

/* avoid using NULL, modern C++ should use nullptr */
```

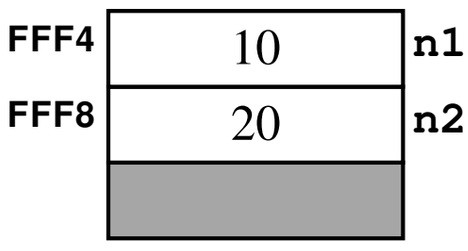
Passing by reference

- The stack frame stores a pointer to the location in the caller at which that value resides

```
int main() {
    int n1 = 20, n2 = 10;
    if (n1 > n2) swap(n1, n2);
    cout << n1 << " " << n2 << endl;
    return 0;
}

void swap(int & x, int & y) {
    int tmp = x;
    x = y;
    y = tmp;
}
```







Passing by reference

- Any changes are made to the target of the pointer, which means changes remain in effect after the function returns
- Equivalent implementation without address-of operator

```
void swap(int *px, int *py) {
    int tmp = *px;
    *px = *py;
    *py = tmp;
}

// function should be called like this
swap(&n1, &n2);
```

Arrays

- are low-level collections of individual data values
- are countable (count collection size)
- are homogeneous (same data type)
- Limitations:
  - Have a fixed, unchangeable size
  - Offer no support for inserting/deleting elements
  - There is no bound-choking procedure

Array declaration

```
const int N_ELEMS = 1024;

// array declaration: type name[size];
int arrayI[10], intArray[N_ELEMS];

for (int i = 0; i < N_ELEMS; i++) {
    // array selection
    intArray[i] = 10 * i;
    cout << i << " " << intArray[i] << endl;
}

// static initialization
const int DIGITS[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

intArray

0	10	20	30	40	50	60	70	80	90
0	1	2	3	4	5	6	7	8	9

Pointers and arrays

- Array is synonymous with a pointer to its initial element

```
void sort(int array[], int n) {
    for (int lh = 0; lh < n; lh++) {
        int rh = lh;

        for (int i = lh + 1; i < n; i++)
            if (array[i] < array[rh]) rh = i;

        swap(array[lh], array[rh]);
    }
}

void sort(int *array, int n) {
    // ...
}
```

Pointer arithmetic

- Operators ‘+’ and ‘−‘ can be applied to pointers

```
int array[10];
int *p, *q, k = 6;

// p and q point to the 1st address of array
p = array;
q = &array[0];

// p1 and p2 are equivalent
int *p1;
p1 = p + k;
int *p2 = &array[k];

// increment and decrement of pointers
*p1--; // how to check what is this doing?
*p1++; // and that
```

Dynamic memory allocation

- Algorithms and data structures

Styles of memory allocation

- There are three basic styles of allocation
- Static: Global variables

Styles of memory allocation

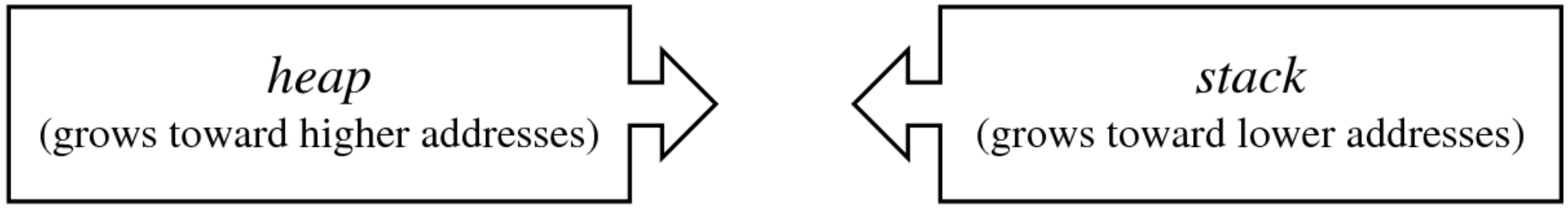
- There are three basic styles of allocation
- Static: Global variables
- Automatic: Local variables inside function

Styles of memory allocation

- There are three basic styles of allocation
  - Static: Global variables
  - Automatic: Local variables inside function
- Dynamic: Require memory space as program runs
  - Allocate and free memory
  - Takes place in the heap (pool of available memory)

Dynamic allocation and the heap

- C++ allows to allocate some of the unused storage to the program whenever your application needs more memory
- Example: If you need an array while the program is running, you can reserve part of the unallocated memory, leaving the rest for subsequent allocations
- The pool of unallocated memory available to a program is called the heap



The new operator

- is the way to allocate memory from the heap
- returns the address of a storage location in the heap
- once allocated in the heap, one can refer to that variable

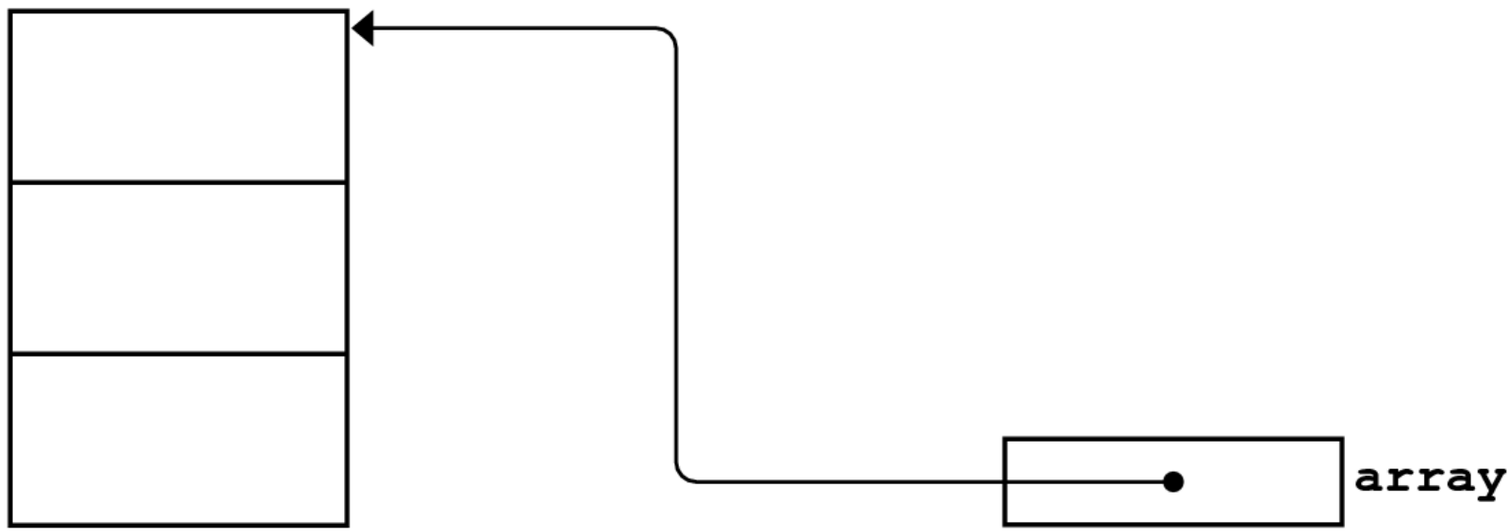
```
int *ip = new int;
*ip = 42;
```

- can be used to allocate objects and *dynamic arrays* on the heap

Dynamic arrays

- is an array allocated in the heap
- To allocate them use the following syntax

```
double *array = new double[3];
a[0] = a[1] = a[2] = 0.0;
```



The delete operator

- takes a pointer allocated by new and frees the memory associated with that pointer
- If the heap memory is an array, you need to add square brackets after the delete
- Some languages support automatic memory freeing that is no longer in active use (*garbage collection*)

```
delete ip; // free memory occupied
delete[] array; // free allocated memory
```

Dynamic memory in C

- new/delete are not available in C
- Use instead `malloc`, `calloc`, `realloc` and `free`
- Defined in header file `<cstdlib>`
- These are also available in C++

Dynamic memory in C: malloc

```
1 void* malloc (size_t size);
```

- Allocates a block of size bytes of memory, returning a pointer to the beginning of the block

```
1 char *buffer = (char*) malloc(i + 1);
2 for (n = 0; n < i; n++)
3     buffer[n] = rand() % 26 + 'a';
4 buffer[i]='\0';
5
6 cout << "Random msg: " << buffer << endl;
7 free (buffer);
```

Dynamic memory in C: calloc

```
1 void* calloc (size_t num, size_t size);
```

- Allocates a block of memory for an array of `num` elements, each of them `size` bytes long, and initializes it to zero

```
1 const size_t i = 123;
2 int *pd = (int*) calloc (i, sizeof(int));
3 for (int n = 0; n < i; n++) {
4     cout << "Enter␣number:␣";
5     cin >> pd[n];
6 }
```

Dynamic memory in C: free

```
1 void free (void* ptr);
```

- A block of memory previously allocated by a call to `malloc`, `calloc or realloc` is deallocated, making it available again

```
1 int *b1, *b2, *b3;
2 b1 = (int*) malloc(100*sizeof(int));
3 b2 = (int*) calloc(100,sizeof(int));
4 b3 = (int*) realloc(b2, 500*sizeof(int));
5 free(b1);
6 free(b3);
```

Outlook

- C++ is a complex language with a huge and ever-growing standard library
  - ✤ Don't be overwhelmed
  - ✤ Learn step by step
- Always use a project that's useful in your life to learn a new programming language

Thank you!

Bibliography

A. Eric Roberts, *Programming abstractions in C++* (2013)

B. http://www.cplusplus.com/

C. https://en.cppreference.com/

D. Scott Meyers, *Effective modern C++* (2015)