Part 3: C/C++ Basics Continued

Contents

[DOCUMENT FINALIZED]

C++ : feature available in C++ but not in C

C Structures

Week 4

```
struct Point {    // a point consists of
  int x, y;       // two int coordinates
};                // ; required

// In C++ you can omit struct when
// defining variables, it is redundant.
// Such redundancies will be highlighted:
struct Point p;   // define a point

p.x = 10; // set x component of p to 10
p.y = 20; // set y component of p to 20
```

- structure variables are collections of variables, grouped together under a single name
- structs help organize data
- Java classes are similar

Structure Definition

```
struct PersonInfo {
  int height;
  int weight;
  Date birthday;
};
```

```
struct PersonInfo x;

x.height         = 180;
x.weight         = 80;
x.birthday.year  = 1965;
x.birthday.month = 4;
x.birthday.day   = 5;
```

- Data components are accessed by the . operator
- Structure components are laid out sequentially in memory
- Recursive structure definitions are not allowed, i.e., structure Foo can't have a component of type Foo.

Structure Initialization

```
struct Date {
  int year;
  int month;
  int day;
};

struct Date date = { 1965, 4, 5 };
```

- Structure components are not initialized by default!
- To initialize on the spot add

  `= { <exp>, <exp>, ... <exp> }`

  where <exp> is an expression.
- Data members are initialized corresponding to their order in definition
- If fewer expressions are listed than there are variables, the remaining components are set to 0

  Quick initialization of all components with 0:

  `Date date = { 0 };`

Structure Assignment

```
struct Point { int x, y; };

struct Point p1, p2;

p1 = p2;
// same as p1.x = p2.x; p1.y = p2.y;
```

- Structure variables can occur on the left hand side of assignments
- Type of the right hand side expression must match
- All structure components are copied one by one
- In C, structures can't be compared, i.e.

  `if (p1 < p2) { ... }`

  is illegal in C

  However, in C++ operators can be overloaded so that they work with structs and code like above works. Search for operator overloading in C++ to learn more. We will not cover this topic in this course.

Structures and Functions

```
struct Point { int x, y; };

struct Point times2(struct Point p)
{
  p.x *= 2;  // doubling local variable p
  p.y *= 2;
  return p;  // return local variable to caller
}

struct Point p, q;
// pass p by value and store result in q
q = times2(p);

// copying can be costly if structures are big

// faster C++ code using reference
void times2(Point &p)
{
  p.x *= 2;
  p.y *= 2;
}

Point p;
times2(p); // p is doubled, no copying
```

Structures and Functions (Continued)

- Structures can be passed to functions by value or by reference
- Passing by reference is faster because nothing has to be copied
- Returning structures is allowed.
- Difference to Java: structures in C are stored on the stack unless dynamically allocated from heap memory. We will soon see how this works.

Structure Memory Layout

```
struct Foo {
  char a; int b; char c;
} x;

How x is stored in memory:
  x.a     1 byte
  x.b     4 bytes
  x.c     1 byte
          total 6 bytes
```

```
struct Bar {
  char a; char c; int b;
} y;

How y is stored in mem.:
  y.a     1 byte
  y.b     1 byte
  y.c     4 bytes
          total 6 bytes
```

- In general, structure components are stored in consecutive memory locations.
- However, the exact memory layout and size of structures depend on compiler and machine architecture. “Holes” may be introduced to trade space for access speed ...

Structure Memory Layout (Continued)

```
struct Foo {
  char a; int b; char c;
} x;

How x is really stored
in memory:
  x.a     1 byte
  unused 3 bytes
  x.b     4 bytes
  x.c     1 byte
  unused 3 bytes   total 12
```

```
struct Bar {
  char a; char c; int b;
} y;

How y is really stored
in memory:
  y.a     1 byte
  y.b     1 byte
  unused 2 bytes
  y.c     4 bytes total 8
```

For instance, in g++ under Linux for Intel/AMD x86 CPUs:

- ints are aligned to addresses divisible by 4
- shorts are aligned to addresses divisible by 2

Structure Memory Layout (Continued)

```
Physical memory organization: 4-byte words

| 0  1  2  3 |  int stored at 0..3: 1 access
| 4  5  6  7 |  int stored at 5..8: 2 accesses!
| 8  9 10 11 |
|12 13 14 15 |  = word
...
```

- Accessing aligned ints is faster than unaligned ints
- Reason: memory is organized as a sequence of words which usually contain 4, 8, or even 16 bytes, and the CPU reads and stores that number of bytes whenever it accesses memory.
  - aligned int: just one memory access
  - unaligned int: two accesses

Packed Structures in gcc/g++

```
struct Foo {
  char a; int b; char c;
} x;

How x is stored in memory:
  x.a     1 byte
  unused 3 bytes
  x.b     4 bytes
  x.c     1 byte
  unused 3 bytes total 12
```

```
struct
__attribute__((packed)) Foo
{
  char a; int b; char c;
} x;

How x is stored now:
  x.a     1 byte
  x.b     4 bytes
  x.c     1 byte     total 6
```

- Save memory with `__attribute__((packed))`
- packed structures: smaller, but slower access
- non-standard C language extension
- Compiles only with gcc/g++

sizeof Operator

```
struct Point { int x, y; };

sizeof(char) == 1

sizeof(int) == 4

sizeof(double) == 8

int x;
struct Point p;
printf("%d %d\n", sizeof(x), sizeof(p));
// 4 8
```

The sizeof operator can be applied to any type or variable.

It returns the number of bytes a variable occupies in memory and will become important later when we allocate memory dynamically.

Arrays

```
int A[8];      // array A contains 8 ints
double B[100]; // array B contains 100 doubles
struct Point points[50]; // 50 points

// access elements by index, starting with 0
A[0] = 5; // store 5 in first array element
A[7] = 0; // store 0 in last array element

// initialize all values in B with 1.0
for (int i=0; i < 100; i++) B[i] = 1.0;
```

Arrays group together variables of identical type

Define array containing 8 ints: `int A[8];`

Elements can be accessed by index: `A[i] = 0;`

If `i` equals 5, element `A[5]` is set to 0

Array Definition

```
const int N = 5;
const int M = 256;

char A[N];      // 5 characters A[0]..A[4]

int B[M];       // 256 ints B[0]..B[255]

float C[2*M];   // 512 floats C[0]..C[511]
```

- Syntax: <type> <ident> [<integer-expr>] ;
- Integer expression defines the number of objects in the array.
- Array indexes always start with 0
- If an array contains N elements, valid indexes are 0,...,$N-1$
- In C, array elements are not initialized!

Array Initializer Lists

```
int  A[4];       // 4 integers - not initialized!
int  B[4] = { 4, 3, 2, 1 };  // B[0]=4,..B[3]=1
char C[2] = { 'a','b','c' }; // invalid! too many
char D[]  = { 'a','b','c' }; // OK, defines D[3]
int  E[2] = { 1 };           // OK, E[0]=1 E[1]=0
```

- The list of expressions is evaluated and the results are assigned to the array elements in turn
- If list is shorter than the array length, the rest is set to 0
- Array size can be omitted; it is then equal to the list length

Array Element Access

```
#include <assert.h> // make assert macro known
const int N = 10; // good practice to name const.
int A[N];

for (int i=1; i <= N; ++i) {
  printf("%d ", A[i]);
}
// oops! this is a bug which is hard to catch!

for (int i=1; i <= N; ++i) { // still buggy!
  // aborts program if i is out of range
  assert(i >= 0 && i < N);
  printf("%d ", A[i]);
}
```

- Syntax: <ident> [<integer-expression>]
- The expression is evaluated and the array element with that index is accessed
- It is a serious logical error if a program accesses elements outside the valid index range.
- The C runtime system doesn't check index validity!

Array Memory Layout

```
int A[8];

x = address of first byte of A in memory

address     content
x   ..x+3   A[0]
x+4 ..x+7   A[1]
x+8 ..x+11  A[2]
x+12..x+15  A[3]
x+16..x+19  A[4]
x+20..x+23  A[5]
x+24..x+27  A[6]
x+28..x+31  A[7]
```

Elements are laid out consecutively in memory

This array occupies $8 \cdot \text{sizeof(int)} = 32$ bytes in memory

Arrays as Function Parameters

```
const int N = 10;
int A[N];

void sort(int A[]) { ... } // doesn't work!
// function sort has no access to A's size

void sort(int A[], int size) { ... } // OK

sort(A, N);  // OK
```

- In C, arrays are passed by reference
- Parameter syntax: <type> <ident> [ ]
- An array parameter is a reference pointing to the first array element.
- I.e., arrays are not copied into local variables when passed to functions and changes will be applied to the array in the caller.
- There is no size information attached to array parameters! Need to pass number of elements explicitly.
- Functions cannot return arrays.

Programming with Arrays

- Most computation cycles are spent on searching and sorting
- Need to be implemented efficiently
- Details in algorithms/data structure courses such as CMPUT 204
- Here, we only cover some basics to illustrate C/C++ programming with arrays:
  - linear search
  - naive sorting

Search 1

- Task: find an element in an array
- If found, return smallest index of matching elements, otherwise return -1

```
// pre-condition: A has at least size elements
// post-condition: returned value is smallest
// index of e in A[0..size-1], or -1 if not found
int find(int e, const int A[], int size)
{
  for (int i=0; i < size; ++i) {
    if (A[i] == e) {
      return i;
    }
  }
  return -1;
}

int A[5] = { 5, 4, 3, 2, 1 };
printf("%d", find(2, A, 5)); // prints 3
```

const guards against accidentally changing array elements. E.g., `A[0] = 0;` would be illegal.

Search 2

Task: return index of maximum array element

```
// pre-condition: A has at least size > 0 elems.
// post-condition: returned value is the index
// of the maximum element in A[0..size-1]
int indexOfMax(const int A[], int size)
{
  assert(size > 0);   // check pre-condition
  int max_ind = 0;    // current index of max.
  int max_val = A[0]; // current maximum value

  for (int i=1; i < size; ++i) {
    if (A[i] > max_val) {
      // found a bigger element => update
      max_val = A[i]; max_ind = i;
    }
  }
  return max_ind;
}

int A[5] = { 5, 4, 3, 2, 1 };
printf("%d", indexOfMax(A, 5)); // prints 0
```

Sorting

Task: sort an array in non-decreasing order

Idea: find maximal element, swap it with the last element, and apply the same algorithm to the remaining array part. This sorting algorithm is called "Selection Sort".

```
// pre-condition: A has at least size elements
// post-condition: A[0] <= A[1] <=...<= A[size-1]
void sort(int A[], int size)
{
  for (int l=size; l > 1; --l) {
    // swap max. element in A[0..l-1] with A[l-1]
    swap(A[indexOfMax(A, l)], A[l-1]);
  }
}

int A[5] = { 5, 4, 3, 2, 1 };
sort(A, 5);
// A now 1 2 3 4 5
```

Pointers

```
int *p;     // p is a pointer to an int variable
int a;

p = &a;     // the address of a is assigned to p
            // p points to a now
int *q = p; // q now also points to a
```

- Pointers are variables that contain the address of a variable
- A leading * in a variable definition indicates a pointer variable; no default initialization!
- In pointer assignments the & (address) operator is used to determine the address of an object in memory (first byte)
- 0 is a special pointer value: it can be assigned to any pointer variable regardless of its type.
- Memory at address 0 is not part of your process memory. Can indicate no memory, invalid pointer, no successor, etc.

Dereferencing Pointers

```
int x = 1, y;
int *ip;    // ip is a pointer to int, or:
            // "the object ip points to is an int"
            // unitialized!
ip = &x;    // ip now points to x
y = *ip;    // y is now 1
*ip = 0;    // x is now 0
*ip += 10;  // increments x by 10
```

- The unary operator * is used for indirection (a.k.a. dereferencing)
- When applied to a pointer the result represents the variable the pointer points to.

Operators & *

```
short x = 5;
short *ip = &x;    // a pointer to x
short y = *ip + 1; // takes whatever ip points
                   // to (x), adds 1 and assigns
                   // the result (6) to y

(*ip)++; // increments what ip points to (x)
++*ip;   // ditto

*ip++;   // increments ip; * has no effect here
short y = *ip++; // this copies the variable ip
             // points to to y, and increments ip
```

- Higher precedence than arithmetic operators
- Same precedence as ++ -- (rtl associativity)
- Sometimes parentheses are needed

Pointers and Arrays

- In C there is a strong relationship between pointers and arrays
- Any [ ] operation can be expressed by an equivalent pointer expression (see below)
- The pointer version used to be faster, but is harder to understand
- Modern compilers generate equally fast code
- Arrays are passed to functions as a pointer to the first element ⇝ size information is lost

Pointers and Arrays (Continued)

- pa+C points to the C-th successor of *pa
- pa-C points to the C-th predecessor of *pa
- The actual address is incremented resp. decremented by sizeof(*pa) * C

  E.g. by 4*C if pa is an int pointer
- Array definitions ⇝ constant pointers

```
int A[10], *pa;
pa = A; // legal
A = pa; // illegal
```

- a[i] equivalent to *(a+i). Why?
- &a[i] equivalent to a+i. Why?

Array Access Example

```
int A[4];
int *pa = &A[0];  // or:  = A;  equivalent

  | A[0]  |  A[1]  |  A[2]  |  A[3] |

   ^        ^        ^        ^
   pa       pa+1     pa+2     pa+3


*pa     = 1;  // sets A[0] = 1
*(pa+1) = 2;  // sets A[1] = 2
*(pa+2) = 3;  // sets A[2] = 3
*(pa+3) = 4;  // sets A[3] = 4
```

Pointer Arithmetic

```
int n;
T *p; ...
p += n; // increments p by n*sizeof(T)
p -= n; // decrements p by n*sizeof(T)
```

- If p and q point to elements in the same array, == != < > <= >= between p and q work properly
- Pointer subtraction also valid: if p and q point to elements of the same array and p >= q, then p-q is the number of elements from p to q exclusive.
- All other pointer operations are illegal
- The runtime system does not check whether pointers actually point to variables of the correct type or even at an object under your control. But it will end your process with a "segmentation fault" if you try to change memory cells your process doesn't own. E.g.

```
int *p = 0; // address 0 never yours
*p = 5;     // this kills your process
```

Pointers and Structures

```
struct Point { int x, y; } p1, p2, *pp = &p2;

// store p1.x into the x component of
// the point pp points to (p2)
pp->x = p1.x;
// same for y
pp->y = p1.y;

(*pp).x = p1.x; // equivalent
(*pp).y = p1.y; // *pp = point pp points to

*pp = p1; // stores both x and y, same as p2 = p1
```

Two equivalent ways to access structure members via pointers:

- (*p).member
- p->member

Pointer Arrays, Pointer to Pointers

```
int *A[4]; // array of 4 pointers to int
int a, b, c, d;
A[0] = &a; // store addresses of a b c d in A
A[1] = &b;
A[2] = &c;
A[3] = &d;

*A[2] = 5;                     // same as c = 5;
*A[3] = *A[0] + *A[1] + *A[2]; // d = a + b + c;

int **p; // p is a pointer to a pointer to an int
         // same as
int *p[];// p points to array of int pointers

p = A;
**p = 0;      // same as a = 0;
**(p+1) = 1; // same as b = 1;
```

Pointers are variables themselves, thus

- they can be stored in arrays, and
- they can point to pointers, in which case we need more than one * to access values

Pointers vs. References

```
struct Point { int x, y; };

// reference version C++
void foo(Point &p)
{
  p.x = p.y = 0;
}

// pointer version
void bar(struct Point *p)
{
  p->x = p->y = 0;
}
```

References are internally represented as pointers.

Both functions do the same thing and are equally fast.

Main difference: references always refer to an existing variable, whereas pointers can point to any address in memory (including 0, which often has a special meaning, such as end-of-list).

C Input

```
#include <stdio.h>

int c = getchar();    // read one byte from stdin
if (c == EOF) {       // end of file or error?
  if (feof(stdin)) {
    ...               // end of file
  } else {
    ...               // error
  }
} else {
  printf("%c", c);    // c valid, process it
}
```

To write more useful programs, we want to read data from files, pipes, or the keyboard, i.e., from the standard input stream.

getchar() is a C library function that reads one character from standard input.

It returns a larger data type (int) because it needs to be able to signal events such as end-of-file reached or read error to the caller.

This is accomplished by returning special value EOF.

C Input (Continued)

If a subsequent call of feof(stdin) returns true, the end of the input is reached, otherwise a read error occurred.

Use man getchar to learn more. We will cover input/output functions in more detail later.

Another useful function is scanf(), which reads words from the input, converts them and stores the results in variables whose addresses are passed as parameters.

scanf() returns EOF if the end of file is reached or a read error occurred. Otherwise, it will return the number of variables that were successfully read.

scanf() uses format strings similar to those understood by printf().

Consult man scanf to learn more.

C Input (Continued)

```
// read integers from standard input and sum them up

#include <stdio.h>

int main()
{
  int sum = 0;              // running total
  int res;                  // scanf return value

  for (;;) {                // "forever"
    int x;
    res = scanf("%d", &x);  // read integer from stdin
                            // store it in x
    if (res != 1) {
      break;                // read error or EOF
    }
    sum += x;
  }

  if (res == 0 || (res == EOF && !feof(stdin))) {
    printf("corrupt input\n");
    return 1;               // something wrong
  }

  printf("%d\n", sum);
  return 0;                 // all went well
}
```

C Input (Continued)

```
Sample input and output:

file foo contains:
1  2
  3  4
 5

./a.out < foo
=> 15

./a.out followed by typing this on keyboard:
1
2
3.5
ctrl-d
=> corrupt input


ascanf skips spaces and newline characters!
Thus, the input format is irrelevant.
```

C-Strings

Week 5

C-strings are sequences of characters

C-string constants are double-quoted

```
  "I am a string"
  "hello world\n"
```

- Can contain escape sequences such as `\n` `\a` `\t`
- " in the text is represented by `\"`
- \ in the text is represented by `\\`

  e.g. `"\"quoted text\""`

You have seen them before when using `printf`:

```
printf("hello world");
printf("value=%d\n", x);
```

C-String Representation

```
char s[9] = "Hello!";

s[0] s[1] s[2] s[3] s[4] s[5] s[6] s[7] s[8]
  H    e    l    l    o    !   \0  both undef

char s[] = "Hello!"; // reserves enough space to
                     // hold Hello! + \0
-> sizeof(s) = 7
```

- Array of characters which contains the character sequence plus end-marker `'\0'` (0 byte)
- C-strings can be initialized via =
- Some operations are inefficient because like any other array, when passed to functions as parameters the length is lost.
- C++ comes with a more sophisticated string class

C-String Pitfalls

- Ensure that the char array is big enough - must hold characters + end-marker 0
- Character with code 0 cannot be represented in a C-string because 0 indicates end-of-string
- Assignments other than initializations are illegal
- == and other relational operators such as >= don't work with C-strings
- Does not sound very useful
- Solution: library functions

C-String Library Functions

To make compiler aware of these functions use
`#include <string.h>`

`int strlen(const char s[]);`

- returns the number of characters in s excluding the end-marker

`void strcpy(char dst[], const char src[]);`

- copies string src to dst
  (dst must be big enough!)

`void strcat(char dst[], const char src[]);`

- appends string src to dst overwriting its end-marker and adds '\0'
  (dst must be big enough!)

C-String Library Functions Continued

`int strcmp(const char s1[], const char s2[]);`

- compares strings `s1` and `s2`
- returns 0 if and only if (iff) they are equal
- return value > 0 iff `s1` > `s2` (lexicographical order)
- return value < 0 iff `s1` < `s2`

To learn more about functions in <string.h> issue

`man string.h`

C-String Assignment

```
#include <string.h>

char s1[] = "hello";
char s2[100];

s2 = s1;      //illegal! arrays can't be assigned
strcpy(s2, s1);  // OK, s2 starts with hello\0

char s_too_short[2];

strcpy(s_too_short, s1);
// Undefined! Writes passed end of s_too_short
// Also known as buffer overflow error
// Can lead to all sorts of problems
```

C-String Comparison

```
char a[] = "aaa";
char b[] = "aaaa";
char c[] = "b";
char d[] = "aaa";

a == d   // Probably doesn't do what you'd
         // expect. Compares the addresses of
         // of a and d, not their content!
         // Although in this case the content
         // is identical, the result is false
         // because a, d are stored in different
         // memory locations.

a < b    // is a located ahead of b ?
a > b    // is a located after b ?

strcmp(a, a) == 0  // true, strings identical
strcmp(a, b) < 0   // true, a lex. less than b
strcmp(c, b) > 0   // true, c lex. greater than b

strlen(b) == 4     // true
```

strlen & strcpy Implementation

```
// return length of string, pointer version
int strlen(const char *s)
{
  const char *p = s;
  while (*p) ++p;
  return p-s; // pointer arithmetic
}


// copy t to s, pointer version
void strcpy(char *dst, const char *src)
{
  while (*dst++ = *src++);
}
```

How do these functions work?

strcat Implementation

```
// appends the src string to the dst string
// overwriting the '\0' character at the end of
// dst and then adds a terminating '\0' character
void strcat(char dst[], const char src[])
{
  int i=0;
  while (dst[i]) ++i; // find end-marker
  int j=0;
  char c;
  do {
    c = dst[i++] = src[j++]; // append src
  } while (c);
}
```

How does this function work?

C-String Output

```
char hw[] = "hello world";

// %s indicates string in printf
// format strings
printf("the string is %s", hw);
// prints: the string is hello world
```

Format string %s has many options.

Run `man printf` to learn more.

C-String Input

```
char input[20];

int status = scanf("%19s", input);
// reads the next word into input
// (at most 19 characters)
```

When reading strings, scanf skips "white space" such as space, tab, and newline characters and stores the following word, which ends with the next white space or when EOF is reached, into the string that is passed as a parameter.

When providing length n after %, no more than n characters are stored in the string variable, whose size has to be at least $n + 1$.

Always use a length restriction! Otherwise, scanf may write what it reads passed the end of the array!

More details, including a description of the return value of `scanf`, can be found with man scanf.

Reading Entire Lines

```
const int N = 100;
char input[N];

// stores line until reaching \n into input
// (at most N-1 chars)
int result = fgets(input, N, stdin);
// result == 0 <=> EOF or read error
```

Safe way of reading lines from standard input.

```
// simple line counting
char line[100];
int count = 0;
while (fgets(line, sizeof(line), fp)) {
  count++;
}
```

Command Line Parameters

```
// print all command line arguments
#include <stdio.h>

int main(int argc, char *argv[])
{
  for (int i=0; i < argc; ++i) {
    printf("argument %d : %s\n", i, argv[i]);
  }
}
```

prototype: `int main(int argc, char *argv[]);`

- argc: number of command line arguments
- argv: array of pointers to command line args.
- argv[0]: pointer to program name string
- argv[1]: pointer to first argument string, ...

Example

```
./foo -o x "foo bar" 'moe and hal'

output:

argument 0 : ./foo
argument 1 : -o
argument 2 : x
argument 3 : foo bar
argument 4 : moe and hal
```

- When invoking a command, the shell cuts the input line into pieces
- Uses space as delimiter (but obeys strings enclosed in " and ')
- Removes leading and trailing spaces for arguments not enclosed by "'

typedef

```
typedef signed char   sint1;
typedef unsigned char uint1;
typedef signed int    sint4;
typedef float         real4;
typedef double        real8;
sint4 i;  // signed four-byte integer
uint1 c;  // unsigned one-byte integer
real4 r;  // float
real8 d;  // double

typedef const char *ccptr;
int strlen(ccptr s) { ... }
```

- Type aliases are new type names for existing types
- Syntax: typedef <variable-definition>;
- Variable identifier is treated as type name
- Increases readability and portability
- Can simplify complex type expressions

typedef (Continued)

Another usage of `typedef` is to get around C's slightly annoying requirement to add the redundant keyword `struct` when struct variables are used.

```
struct Point { int x, y; };

struct Point u; // C insists

typedef struct { int x, y; } Point;

Point v; // aha!
```

Dynamic Memory Allocation in C++

- Local variables and functions parameters are located on the stack (LIFO data structure) as we have seen before.
- Dynamic memory is allocated from a different part of memory called heap.
- In C++, operator `new` dynamically allocates memory and operator `delete` is used to release memory when it is no longer needed. This can be done later, even in a different function.
- In C, functions `malloc` and `free` are used to allocate and release memory.
- YOU need to decide when memory is no longer used. The compiler doesn't know.
- The runtime system could know, but C and C++ don't support garbage collection (yet), unlike Java and Python.

Operator new C++

```
int *p = new int;    // allocates space
                     // for one int on the heap
                     // p now points to it

*p = 1;   // use allocated variable: set it to 1
```

- Syntax: `new <type>`
- Allocates space for a variable of type <type> on the heap and returns a pointer to it.
- No initialization if <type> is a C data type, such as int or double.
- If <type> is a struct, its constructor is called that initializes the object (we'll come to that later).
- If no memory is available your program is terminated (Well, this strictly isn't the whole truth, but good enough for our purposes in CMPUT 201).

Operator delete C++

```
// allocate new int variable on heap
int *p = new int;

// work with variable
*p = 0;
(*p)++; ...

// free variable when it is no longer needed
delete p;

// accessing *p after deleting it is a SERIOUS
// logical error, which is hard to track.
*p = 0; // OUCH, but program resumes

// to help detecting such cases, setting p = 0
// right after delete is a good idea
delete p;
p = 0;  // safeguard

*p = 0;
// now kills the process, because it tries
// to write to a memory location it doesn't own
```

Operator delete C++ (Continued)

- Frees memory when it is no longer needed.
- Calls destructor for structs (later).
- Syntax: `delete <pointer-to-allocated-mem>`
- Good practice: set pointer to 0 after delete to prevent further access of this address through this pointer.
- Also: when designing your program make sure each heap object has exactly one owner who is responsible for its deletion. It is a serious error if `free` is called twice on one object.

Dynamic Arrays

```
// allocate array of 100 float variables
float *p = new float[100];
// initialize all values
for (int i=0; i < 100; ++i) p[i] = 0.0;
...
// free memory when array is no longer used
delete [] p;
p = 0;  // safeguard
```

- Syntax: `new <type>[<num-of-elements>]`
- Allocates an array of elements of type `<type>`.
- Elements are not initialized for basic C types.
- When no longer used free memory with `delete [ ] <pointer-to-dynamic-array>`

new/delete Match

- `new`/`delete` come in pairs: for every `new` there should be at least one `delete` in your program
- More specifically:
  - For every `new` at least one corresponding `delete`
  - For every `new[]` at least one corresponding `delete[]`
- When mixed, like so:

```
int *p = new int;
delete [] p; // ERROR
```

the computation result is undefined.

Speed / Memory Issues

- Allocating dynamic memory is SLOW compared to stack-based allocation.
- The operating system has to maintain list of available memory blocks.
- If speed is important try to minimize `new`/`delete`. E.g. by pre-allocating and reusing arrays
- `new` allocates more memory than you think (overhead usually 4 or 8 bytes per call).
- Allocating arrays is therefore more efficient than single variables.

Dynamic Memory Allocation in C

```
#include <stdlib.h> // makes malloc/free known

// allocate an array of 100 floats
float *p = malloc(100*sizeof(float));
if (!p) { exit(1); } // out of memory
...
// initialize array
for (int i=0; i < 100; ++i) p[i] = 0.0;
...
// free memory when array is no longer used
free(p);
p = 0;  // safeguard
```

- There are no new/delete operators in C
- Use library function calls instead

  `void *malloc(size_t n);` : allocates n bytes

  `void free(void *p);` : releases memory p points to
- To allocate an array with N elements of type T, you need to pass the size it occupies in memory to `malloc`:

  `malloc(N * sizeof(T))`

Dynamic Memory Allocation in C (Continued)

- `malloc` returns a generic pointer (`void*`) which in C can be assigned to any pointer variable:

  `T *p = malloc(N * sizeof(T));`
- If you want to use `malloc` in C++ programs, the result needs to be cast to the type you want:

  `T *p = (T*) malloc(N * sizeof(T));`

  because C++ doesn't permit assigning `void*` pointers to other pointer types.
- If enough memory was available, `malloc` returns a pointer pointing to the first byte of the allocated memory block. Otherwise, it returns 0.
- When no longer needed, memory blocks need to be returned to the operating system to avoid running out of memory. This can be accomplished by calling

  `free(p);`

  where p points to a memory block that was allocated by `malloc`.
- To learn more about `malloc/free`, run man malloc

Memory Allocation Example

C++ Version:

```
struct Point { int x, y; };

const int N = 1000000;

// allocate array of 1000000 points
Point *points = new Point[N];

// initialize them
for (int i=0; i < N; i++) {
  points[i].x = points[i].y = 0;
}
... do something else with points

// points no longer needed
// (note [ ] matching [ ] in new)
delete [ ] points;
```

Memory Allocation Example

C Version (compile with gcc -std=c99 ...)

```
#include <stdlib.h>

struct Point { int x, y; };

const int N = 1000000;

// allocate array of 1000000 points
struct Point *points =
      malloc(N * sizeof(struct Point));

// initialize them
for (int i=0; i < N; i++) {
  points[i].x = points[i].y = 0;
}
... do something else with points

// points no longer needed
free(points);
```

Dynamic Memory Allocation in CMPUT 201

In this course we will use the C++ `new`/`delete` operators for dynamic memory allocation (after some exercises practicing malloc/free).

`new`/`delete` are less error-prone and easier to understand. Moreover, they call constructors and destructors automatically, which will be discussed towards the end of the course.

As a consequence, all programs using dynamic memory allocation will have to be compiled with g++.