## Important definitions

**Memory:**

**A sequentially numbered (very large) collection of bytes. The only C/C++ data-type with a size equal to one byte is char. Thus char is sometimes used to represent "just any byte", and not necessarily a character.**

**Address:**

**The index (number) of one specific byte in the memory.**

**Variable (recollect first lecture):**

**The name of one location in memory, storing a value of one specific data-type. Data-type determine size (number of bytes the variable use), and interpretation of the bits stored in those bytes. The name determine the address of the bytes in the memory.**

**Pointer:**

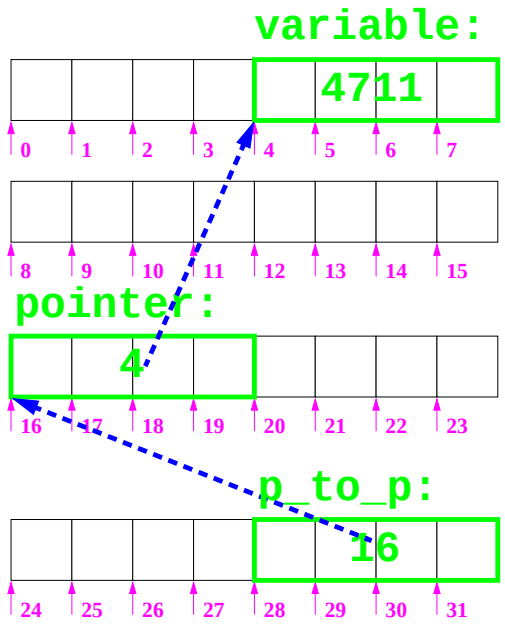
**A variable that store an address.**

**Pointer type:**

**Determine the data-type of the value stored at the address the pointer store. The pointer itself store "just an address".**

## Graphical view

```
int variable = 4711;
int* pointer = &variable;
int** p_to_p = &pointer;
// A 32-byte memory with addresses in red and
// three green variables, two of them pointers
```




## Null-pointer

**First a common error: It is possible to create a pointer that do not contain a valid address. Do not do it by mistake:**

```
int* dangerous; // contain undefined address

// error, will often at random go undetected
cout << *dangerous; // undetected error!!
```

**If you do not know the address you should explicitly show this in the code by using the address 0. The address zero is reserved to always be invalid, and are used to distinguish a valid pointer from an invalid.**

```
int* unused = 0; // pointer point "nowhere"

// error, and it will always be detected
// generates "safe" segmentation fault
cout << *unused << endl;
```

**When you store the address zero in a pointer you create a null-pointer. It will not point to anything. Always create all pointers as null-pointers. Then you can check if a pointer is valid.**

```
if (unused != 0) // is the pointer valid?
{
   cout << *unused << endl;
}
```

## Void pointer

**It is possible to create pointers without knowing the pointer data-type. It is up to the programmer to know the type of the data stored at the address the pointer contain.**

**N.B! You will not need to do this. It is both unnecessary and unsafe. You will always know the data-type. If you ever stumble upon a void* it is good to know what it is, but you should never use it yourself.**

```
int my_value;
void* unknown = &my_value;

// error, type stored at adress unknown!
cout << *unknown << endl;

// the programmer must explicitly tell the type
cout << *((int*)unknown) << endl;
```

**This was often used in the C-language to create "generic" data structures capable of storing pointers to any data-type.**

## Array (1 of 4)

**So far we have said that an address is to ONE byte in the memory, or a pointer the address to ONE variable of some type. But this is actually not defined in the language.**

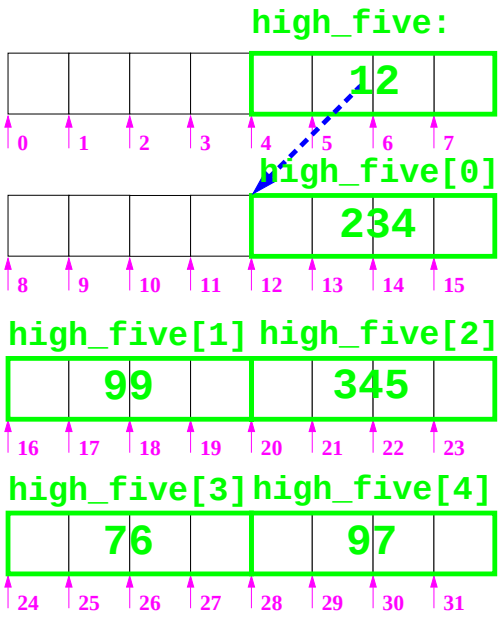
**An address is just the address to the FIRST byte of a long sequence.**

**A pointer contain the address of the FIRST variable in a long sequence.**

**Thus, if the programmer keep track of exactly HOW MANY variables a pointer refer to he/she can use this to build arrays of values.**

```
{
   const int N = 5;

   // create a set of N variables
   // sequentially allocated in memory
   // "array" will be a pointer to the first
   int array[N];

   // initiate each variable immediately
   int high_five[N] = {234,99,345,76,97};

   // or initiate with code
   for (int i = 0; i < N; i = i + 1)
   {
      array[i] = i * i;
   }
} // end of block destroy all
  // variables as usual
```

## Array (2 of 4)




## Array (3 of 4)

**Since an array variable is nothing more than a pointer to a sequence of variables it can be treated as a pointer, and a pointer can be treated as an array.**

**Either way the programmer must keep track of the number of elements pointed to.**

**Since the array already is an address it always passed as an address parameter without using &. At this point it should be clear that address parameters are indeed pointers.**

```
void copy_array(int* destination,
                int* source, int size);
{
   // this will not work, what happen?
   // destination = source;

   // must copy each element instead (works)
   for (int i = 0; i < size; i = i + 1)
   {
      destination[i] = source[i];
   }
}

int main()
{
   int values[] = {1,2,3};
   int copy[3];
   copy_array(copy, values, 3);
   // ...
}
```

## Array (4 of 4)

**To clearly signal that it is an array a function receives as parameter we write the address parameter with array syntax:**

```
// clearly show use of array parameters
void copy_array(int destination[],
                int source[],
                int size);
```

**Since arrays are always passed by address any changes the function do to the elements in the array will be visible in the argument of the calling function.**

**If a function change the arrays passed as parameters it is important for the programmer that call the function to know so. We use the const keyword to clearly tell that a parameter will NOT change.**

```
// clearly show source will not change
void copy_array(int destination[],
                const int source[],
                int size);
```

**This will also prevent the programmer that write the function from doing changes by mistake.**

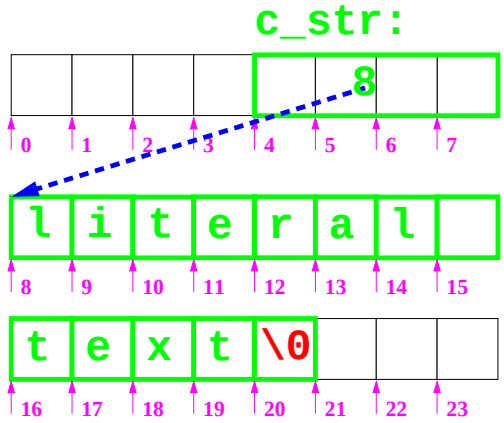
## Character array/pointer

Character pointers, or C-strings are somewhat special. You already know how a C-string are written literally.

```
const char* c_str = "literal text";
```

Literal strings are always const, and instead of keeping track of a separate variable for the length, the final character is the null-character '\0'.

It is not valid to access indexes after the null-character.




## Pointers and Const

When using the const keyword with pointers one question is what is constant. Is it the pointer variable itself, or is it the address pointed to that will not change?

```
// normal const value, a will not change
const int a = 0;
a = 5; // compile error (line 6)

// value at address in x will not change
const int* x;
// value at address in y will not change
int const* y;
// pointer z will not change
int * const z; // compile error (line 13)

// try to change pointer variable
x = &a;
y = &a;
z = &a; // compile error (line 18)

// try to change address pointed to
*x = a; // compile error (line 21)
*y = a; // compile error (line 22)
*z = a;
```

```
gedrix:/home/klaar- g++ const.cc
const.cc: In function 'int main()':
const.cc:6: error: assignment of read-only variable 'a'
const.cc:13: error: uninitialized const 'z'
const.cc:18: error: assignment of read-only variable 'z'
const.cc:18: error: invalid conversion from 'const int*'
to 'int*'
const.cc:21: error: assignment of read-only location
const.cc:22: error: assignment of read-only location
```

## Dynamic memory (1 of 3)

Sometimes the programmer want to create new variables on demand. Possible situations are:

The program should read and store all values from the user until the user decides to stop. It is not known beforehand how many values the user will enter. We can start with an array, but when it is full it can not be extended.

A function that split a sentence into an array of words. It is not known beforehand how large array we need. And even if we define a large enough array it is destroyed when the function returns.

A function that need to insert more data in some dynamic data structure passed as parameter. A local variable can not be used since it is destroyed when the function returns.

Dynamic memory allows the programmer to allocate (reserve) new memory for variables at any point in time. It gives the programmer full control of the memory requirements over time of the program. This comes with a large responsibility: the memory allocated must be released again.

Think of the situation arising if students would always reserve (forever) a new unreserved seat at every lecture, but no students ever released any reservations.

## Dynamic memory (2 of 3)

Attempt one:
We create a pointer initiate it with some address:

```
int* on_demand = 37011; // WRONG
```

Error: We do not know if that address is used by some other variable, program or even by the operating system.

Attempt two:
As we know the operating system keep track of programs and memory we ask the OS to find some free memory:

```
int main()
{
   int* on_demand = new int; // allocate
} // WRONG
```

Error: The memory we received from OS will forevermore be occupied. When we are done with it other programs will want to use it.

Attempt three:

```
int main()
{
   int* on_demand = new int;// allocate
   // ... use the new memory
   delete on_demand; // release or free
   // ... remainder of program
}
```

## Dynamic memory (3 of 3)

**We can allocate arrays dynamically:**

```
int main()
{
   int* array = new int[10];

   // use the array

   delete[] array;

   // remainder of program
}
```

**In this case the above program is identical to the following constant allocated array. But note that the above version can release the allocated memory at any point in time, and allocate a new larger array. The programmer control when to release the memory. In constant allocated version below the compiler takes care of de-allocating. The programmer has only limited control.**

```
int main()
{
   {
      int array[10];

      // use the array

   } // automatically deleted at end of block

   // remainder of program
}
```

## Pointer operations overview

**<> and the text inside should be replaced with the data indicated by the text inside <>.**

```
<data-type>* // pointer declaration
<data-type>& // reference declaration
*<pointer>   // content of address
&<variable>  // address of variable

// access one index in an array
<pointer>[<index-expression>]
*(<pointer> + <index-expression>)

// create variable on demand, will not be
// destroyed automatically at end of block
// must be destroyed manually by calling
// delete before the end of the program
// the result is a pointer to the variable
new <data-type>

// delete a variable created on demand
// may not be used on other variables
// must be done exactly once for each
// variable created on demand
delete <on-demand-variable-pointer>

// on demand array (same rules apply)
new <data-type>[<size-expression>]
delete[] <on-demand-array-pointer>

// next lecture
// access one member in structure pointer
<struct-pointer>-><struct-member-variable>
(*<struct-pointer>).<struct-member-variable>
```

## Pointer operations examples

```
int my_int = 5;
int my_int_array[] = {2, 3, 5, 7, 11};

// create a pointer to unchangeable char
const char* p_to_char;
// set to point to array of characters
p_to_char = "C++";

// create a pointer to int
int* p_to_int;
// set to point to variable
p_to_int = &my_int;

// access content (read or write)
*p_to_int        // preferred way
*(p_to_int + 0)
p_to_int[0]

// set to point to same array as my_int_array
p_to_int = my_int_array;

// calculate address of third item
// N.B! automatically jump one variable
// forward, i.e. adds (2 * sizeof(int))
&my_int_array[2]       &p_to_int[2]
(my_int_array + 2)     (p_to_int + 2)

// access content of third item
// N.B! first item is at index 0
my_int_array[2]  // preferred way
*(my_int_array + 2)
p_to_int[2]      // preferred way
*(p_to_int + 2)
```

## Lecture 4b separator page

## Dynamic memory examples

```
// set to point to one new variable
// created on demand (must be deleted again)
p_to_int = new int;

// delete on demand created variable
delete p_to_int;

// set to point to new on demand array
p_to_int = new int[5];

// delete on demand created array
delete[] p_to_int;

// pointer to structure
struct chained_element* first = 0;

// add element to chain
first = new struct chained_element;
// ... initiate other members
first->next = 0;

// insert element first in chain
struct chained_element* to_add;
to_add = new struct chained_element;
// ... initiate other members
to_add->next = first;
first = to_add;
// to_add is now stored in first
// set it to zero to avoid duplicate pointers
// to same memory (can not delete twice!)
to_add = 0;
```

## Struct: A custom variable container (1 of 2)

**Sometimes we have data that are strongly related, but need more than one variable to store. You have previously seen the complex number, consisting of two parts. another example is the date, consisting of three variables; year, month and day. A third is time, consisting of hours minutes and seconds. A fourth would be persons, consisting of first name, last name, phone number, E-mail and possibly more.**

**In C/C++ it is possible to create a custom data-type to collect such information in one unit.**
**N.B! This describes data-types, it does not create variables!**

```
struct complex
{
   int real;
   int imaginary;
}; // semicolon here very important!!

struct date
{
   int year;
   short int month;
   short int day_of_month;
}; // semicolon here very important!!

struct time_of_day
{
   short int hour;
   short int minute;
   short int second;
}; // semicolon here very important!!
```

## Struct: A custom variable container (2 of 2)

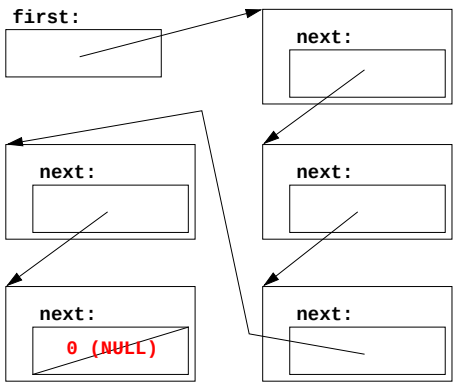
**When the custom struct-types are defined we are ready to create variables of those types:**

```
// create variable
struct date tomorrow;

// initiate the different parts (members)
tomorrow.year = 2009;
tomorrow.month = 10;
tomorrow.day = 6;

// create and initiate variable
struct complex i = {0, 1};

// assign to new variable
struct complex j = i;

// change j to 4711 + i
j.real = 4711;

// do calculations
j = i * j; // ERROR only assignment defined

// create pointer
struct date* oct_six = &tomorrow;

// allocate memory
struct date* my_date = new struct date;

// release memory
delete my_date;
```

## Linked structures

**By using structures together with pointers it is possible to create chains of data in memory. Each element (struct) contain a pointer to one ore more next element(s). Chains, or linked structures are very flexible when it comes to adding and removing new elements. You will create a singly linked list in the lab, by completing the provided code.**

```
struct chain_element
{
   // any data members can be added here
   struct chain_element* next;
};

struct chain_element* first; // and more code
```



## Segmentation fault

**A segmentation fault will, if you are lucky, show up when you handle some pointer wrong. Sometimes it will be called "bus-error" instead.**
**(If you was not careful to use and check for null-pointers, or did not test the program enough this error will not show until you delivered your program to the customer. Then he'll demand money back and buy from someone else.)**
**Thus you are lucky to get this error early. But what to do?**

**First, compile your program with "debug" (-g):**

```
g++ -g my_program.cc
```

**Then you must start the program in the "debugger":**

```
gdb a.out
```

**Inside the debugger, start the program:**

```
run
```

**After it received signal 11, segmentation fault, do:**

```
bt
```

**You will see a trace of which functions was called and at which line the error occurred.**
**To exit the debugger again, type:**

```
q
```

## Debugger example (1 of 2)

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

void bar()
{
   // an invalid pointer to character
   char* invalid = 0;
   // calculate the length of the C-string
   cout << strlen(invalid) << endl;
}

// this function is just to better
// illustrate how a backtrace may look
void foo()
{
   bar();
}

// the execution will eventually
// reach the error in bar
int main()
{
   foo();
}
```

## Debugger example (2 of 2)

**In this example the commands are outlined in blue, and less important program output in smaller font.**

```
gedrix:/home/klaar- g++ -g debug.cc
gedrix:/home/klaar- gdb a.out
GNU gdb 6.5
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public
License, and you are
welcome to change it and/or distribute copies of it under
certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show
warranty" for details.
This GDB was configured as "sparc-sun-solaris2.10"...
(gdb) run
Starting program: /home/klaar/a.out

Program received signal SIGSEGV, Segmentation fault.
0xff0320d0 in strlen () from /lib/libc.so.1
(gdb) bt
#0  0xff0320d0 in strlen () from /lib/libc.so.1
#1  0x00011368 in bar () at debug.cc:7
#2  0x000113b4 in foo () at debug.cc:12
#3  0x000113cc in main () at debug.cc:17
(gdb) q
The program is running.  Exit anyway? (y or n) y
```

**Observe that it may look like the error is in some library code (strlen) you never touched. This happen often, do not let it confuse you. The error is in your code, but it is just detected in some code executed later on.**
**The debugger can do much more than this, but you must learn this on your own, type "help" in the debugger.**