Memory allocation

C functions *malloc()*, *calloc()*, *realloc()* and *free()* are applicable in C++. But when we are working with objects, they are useless.

C++ unary operator *new* allocates memory:

```
pointer = new type[number of elements];
```

If the number of elements is 1, write just:

```
pointer = new type;
```

Examples:

```
double *pd = new double; // 8 bytes
double *pdm = new double[10]; // 10 * 8 bytes
double **ppdm = new double *[10]; // 10 pointers to doubles
```

To release the allocated memory use unary operator *delete*:

```
delete pointer;
```

Example:

```
delete ppdm;
```

You may have heard that if the memory is allocated for an array (i.e. the *new* expression contains brackets), then for releasing operator *delete[]* and not *delete* must be used. The truth is that if the members of array are not objects then operator *delete* is perfectly applicable. About *delete[]* we'll discuss in the next chapter.

Unicode in C++ (1)

ASCII : one byte per character, max 256 different characters (0…255).

Unicode: international standard, each character has its own unique code point marked as U + xxxx (xxxx is a hexadecimal number).

The latest version 10.0 contains 136,765 characters including historic scripts (for example, the Egyptian hieroglyphs) and multiple symbol sets. Max 1,114,112 characters are possible. See https://unicode-table.com/en/

Unicode is implemented by different encodings: UCS-2, UTF-16, UTF-32, UTF-8.

UCS - Universal Character Set, UTF - Unicode Transformation Format.

UCS-2: 2 bytes per character. The first 256 characters match the ASCII codes. Max 65,536 bytes.

UTF-16 (Windows, Java): 2 or 4 bytes per character. Mostly 2 bytes as in UCS-2. If the contents of two-byte field is from interval [0xD800 : 0xD8FF], the code occupies also the next two bytes. The four-byte codes (called as surrogates) are met very seldom.

UTF-32: 4 bytes per character.

UTF-8 (web, Unix, Linux): 1, 2, 3 or 4 bytes per character. If the highest bit of the byte is 0, the code occupies 1 byte, otherwise we have to check the next byte. If the highest bit of the second byte is 0, the code occupies 2 bytes and so on.

Unicode in C++ (2)

C++ has new fundamental type: *wchar_t* (wide character). The standard does not state how many bytes a *wchar_t* variable should occupy and which encoding system must be used.

Character constants:

```
char c = 'A'; // one byte
wchar_t wc = L'A'; // in Windows 2 bytes
```

String constants:

```
const char *pc = "ABC"; // 3 bytes + 1 byte for terminating 0
const wchar_t *pwc = L"ABC"; // in Windows 6 bytes + 2 bytes for terminating 0-s
```

String manipulation (1)

The complete list of string manipulation functions implemented in Visual Studio is on https://docs.microsoft.com/en-us/cpp/c-runtime-library/string-manipulation-crt?view=vs-2017.
The most frequently used:

Task	ASCII	Unicode
String length	strlen	wcslen
Non-secure copy	strcpy	wcscpy
Secure copy	strcpy_s	wcscpy_s
Comparision	strcmp	wcscmp
Find character	strchr	wcschr
Find substring	strstr	wcsstr
Non-secure append	strcat	wcscat
Secure append	strcat_s	wcscat_s
Printing	printf	wprintf

String manipulation (2)

In secure copy and append functions we have to present the length of output buffer:

```
char *pBuf = new char[10];
strcpy(pBuf, "Hello");
strcpy_s(pBuf, 10, "Hello");
wchar_t *pWideBuf  = new wchar_t[10]; // actually 20 bytes
wcscpy_s(pWideBuf, 10, L"Hello");
   // not 20, the length presents the number of units, not the number of bytes
   // write the exact number, otherwise you may corrupt your memory
```

In Visual Studio calls to non-secure functions are handled as errors in code. However, if you still prefer the old-style non-secure functions, open the Visual Studio *Properties* window and add _*CRT_SECURE_NO_WARNINGS* constant to preprocessor definitions. Or write

```
#pragma warning(disable: 4996)
```

at the beginning of your *.cpp file.

Scope resolution operator

```
int iii;      // global
void fun()
{
  int iii;    // local in fun()
  iii = 5;    // to use the global iii write ::iii = 5
  while (1)
 {
    int iii;  // local in while block
    iii = 0; // to use the global iii write ::(::iii) = 0
             // to use the local iii write ::iii = 0
   ………….
 }
 iii = 10;   // to use the global iii write ::iii = 10
 ……..
}
```

Default values for arguments

```
void fun1(double, double, int = 0); // prototype specifies the default values
void fun1(double d1, double d2, int ii) // definition
{
……………………………………….
}
fun1(5.0, 6.0, 1); // call to fun1(), d1 = 5.0, d2 = 6.0, ii = 1
fun1(5.0, 6.0);    // call to fun1(), d1 = 5.0, d2 = 6.0, ii = 0

void fun2(double = 0, double = 0, int = 0); // prototype
void fun2(double d1, double d2, int ii) // definition
{
……………………………………….
}
fun2(); // call to fun2(), d1 = 0, d2 = 0, ii = 0
fun2(5.0);    // call to fun2(), d1 = 5.0, d2 = 0, ii = 0
fun2(5.0, 6.0);    // call to fun2(), d1 = 5.0, d2 = 6.0, ii = 0
fun2(5.0, 6.0, 1); // call to fun2(), d1 = 5.0, d2 = 6.0, ii = 1

void fun(double, double = 0, int); // error, arguments with default values must be at
                                   // the end of list
```

Function overloading

In C++ several function may have the same name but only if their number of parameters and / or types of parameters are different. Example:

```
void fun(int, int);  // 1
void fun(double, double); // 2
void fun(); // 3
```

The compiler determines which function to call from the set of actual parameters (so called overload resolution). Example:

```
fun(5, 6); // 1
fun(5.0, 6.0); // 2
fun(); // 3
fun(5, 6.0);   // error, the compiler cannot select between functions
```

Do not forget, that in C++ a constant containing decimal point is of type *double*. To get type *float* use suffix *F* (for example, *123.4F*). To get type *long double* use suffix *L* (for example, 1*23.4L*). In case of integers suffix *U* sets the type to *unsigned int*, *L* to *long int*, *LU* to *unsigned long int*, *LL* to *long long int*, *LLU* to *unsigned long long int*. The suffix may be in lowercase as well as in uppercase.

Inline functions and macros (1)

If a function is defined as *inline*, the compiler may instead of generating a separate block of code simply directly insert the function body into the body of calling function. This is sensible in case of very short functions because the organizing of call, transfer the parameters and return back to the previous location needs a lot of machine commands – often more than the body of called function itself.

Example:

```
double square(double); // prototype
inline double square(double d) // definition
{
 return d *d;
 }
```

In case of call

```
 double x = square(y); // may be compiled also as x = y * y;
```

Keyword *inline* is just a recommendation: the compiler may use inlining but may also ignore our wish.

Visual Studio inlines short function bodies automatically, even if they are not defined with keyword *inline*.

Inline functions and macros (2)

Inline functions may be replaced by preprocessor functions called macros.

Example:

```
#define SQUARE(a)              (a) * (a)
```

In code:

```
double x = SQUARE(y); // the preprocessor replaces this statement with x = (y) * (y)
```

In calls to macro the preprocessor first replaces the formal parameter (here *a*) with actual parameter (here *y*) and then replaces the complete expression.

Remark that the parenthesis in the body of macro are absolutely necessary:

```
double x = SQUARE(y + 1); // we get x = (y + 1) * (y + 1)
```

But if we define the macro as

```
#define SQUARE(a)              a * a
```

we get x = y +1 * y + 1 or actually we get x = 2 * y + 1

References (1)

A reference is an alias for another variable. Below, *ri* is the alias of *i*:

```
int i, *pi = &i, &ri = i;
```

All the following expressions do the same: write value 10 to the four-byte field having now two names: *i* and *ri*:

```
i = 10;
*pi = 10;
ri = 10;
```

A reference must be initialized when created. It is not possible to force a reference to refer to another target.

```
int i, *pi, &ri;  // error
```

Usage example:

```
void swap(int &rx, int &ry) // the prototype is void swap(int &, int &);
{
  int z = rx; rx = ry;  ry = z;
}
```

Instead of call by value here we call by reference:

```
int a = 5, b = 6;
swap(a, b);
```

Actually, *swap* works with *a* and *b*, temporarily named also as *rx* and *ry*.

References (2)

Alternative solution:

```
void swap(int *px, int *py) // the prototype is void swap(int *, int *);
{
   int z = *px; *px = *py; *py = z;
}
```

Call by value (pointers to *a* and *b* are calculated and assigned to *px* an *py*):

```
int a = 5, b = 6;
swap(&a, &b);
```

swap accesses *a* and *b* undirectly, using pointers *px* and *py*.

Example about reference as return value:

```
int &SetValue(int *p, int i) // the prototype is int& SetValue(int *, int);
{
  return *(p + i);
}
int Buf[] = { 1, 2, 3, 4, 5, 6 };
SetValue(Buf, 5) = 16; // Buf is now { 1, 2, 3, 4, 5, 16 }
```

Reference as return value means that the memory field specified by *return* statement (here *Buf[5]*) gets a temporary alternative name (not shown to us). And to this memory field a new value (here 16) is assigned.

References (3)

Lvalue (locator value or left value) is a an object that has address, i.e. it occupies a memory field and this field can be identified by its name or by pointer.

```
double d = exp(5); // d is lvalue
d + 1 = exp(5); // error, the expression on the left side does not specify an lvalue
```

In assignments the left operand must be an lvalue. A reference may also refer only to an lvalue.

Rvalue is an expression that does not represent an object occupying a memory field that can be identified. In other words, if an expression does not specify an lvalue, then it specifies an rvalue. Remark that constants are also rvalues.

A reference as an alternative name for a memory field that can be identified is also an lvalue. Therefore expression *SetValue(Buf, 5) = 16;* is correct: from *SetValue* we get a reference and the assignment is allowed.

```
int &SetValue(int *p, int i)
{
  // return 5; error – constant is an rvalue, reference to it does not exist
  // return *(p + i) +1; error – the expression does not specify an lvalue
     return *(p + i);
}
```

Exceptions (1)

Exceptions are mechanism for a section of code to notify another section of code about an error or other abnormal situation due to it the normal continuation of program run is not possible. The section of code that encounters the problem throws the exception and the section that has to handle the problem catches the exception. The both sections may or may not be parts of the same function.

Suppose we have the following situation:

```
double x;
………………. // in some way compute value of x
if (x >= 0) {
  ……………. // OK, continue
}
else {
 ………………. // cannot continue, we have to inform the function that called our function
}
```

Possible solutions:

- Our function returns error code or simply 1 in case of success and 0 in case of failure.
- One of arguments of our function is the pointer to error code or reference to error code.
- Our function throws an exception

Exceptions (2)

```
int fun(…….)
{
  double x;
  ………………. // in some way compute value of x
  if (x >= 0)
 {
      …………… // OK, continue
      return 1;
  }
  else
 {
       return 0; // cannot continue, we have to inform the function that called our function
  }
}
```

Usage:

```
 if (!fun(....))
{
    printf("Error\n");
     return;
 }
```

Exceptions (3)

```
double fun(……., int *pError) {
  double x;
  ………………. // in some way compute value of x
  if (x >= 0)  {
        …………… // OK, continue
        *pError = 0; // no errors
        return result;
  }
  else {
        *pError = NEGATIVE_NUMBER; // error code defined somewhere in *.h
         return 0; // or any other senseless value
   }
}
```

Usage:

```
 int error_code;
 double result = fun(……., &error_code);
 if  (error_code) {
    printf("Error %d\n", error_code);
     return;
 }
```

Exceptions (4)

```
double fun(…….)  {
    double x;
    ………………. // in some way compute value of x
    if (x <0)  {
        throw NEGATIVE_NUMBER;  // our function exits here
    }
    …………… // OK, continue
    return result;
}
```

Usage:

```
try { // try block contains code that may throw exceptions
    double result = fun(…….);
}
 catch (int error_code)  { // catch block contains code that processes exceptions
     printf("Error %d\n", error_code);
     return;
 }
 …………………….. // if the exception was not thrown, the catch block is ignored
```

In C++ the exception may be an integer, string presenting the error message, pointer, etc.

Exceptions (5)

Suppose that *fun1()* calls *fun2()* and *fun2()* calls *fun3()*. Suppose also that *fun3()* throws exception but its call in *fun2()* is not enclosed into *try-catch* block. In that case the exception will be rethrown from *fun2()* to *fun1()*. If in *fun1()* the call to *fun2()* is enclosed into *try-catch* block, the *catch* in *fun1()* handles it. If not, the exception will be rethrown to function that has called f*un1()*. Thus the exception moves on by the call chain higher and higher. If the exception has reached *main()* and the *main()* is also not able to process it, the program crashes.

A function may throw several exceptions of different types, for example:

```
double fun(double x)  {
    double y;
    ………………. // in some way computes value of y
    if (y <0)
        throw "Negative numbers";  // our function exits here
     int n;
    …………...... // in some way computes value of n
    if (n > 1024)
        throw n; // our function exits here
    …………….
    return result;
}
```

In that case we need several *catch* blocks.

Exceptions (6)

```
try {
    double result = fun(123.456);
}
catch (const char *pMessage)  {
     printf("%s\n", pMessage);
     return;
}
catch (int wrong_value)  {
     printf("Too long array %d\n", wrong_value);
     return;
}
```

Block

```
catch(…) // three points instead of argument
{
  ……………………………… // processing
}
```

handles any exceptions. If there are several *catch* blocks, the universal *catch* must be the last.