Język C++

część 2

nieobiektowe rozszerzenia / zmiany w języku C++
(cd.)

Jarosław Gramacki
Instytut Informatyki i Elektroniki

- dyrektywa #define
  - dyrektywa **preprocesora** (nie kompilatora) używana **1.** do definiowania stałych symbolicznych
  - **dyrektywa** `#define` używana jest również **2**. do definiowania funkcji w celu zwiększenia szybkości działania programu (kod takiej funkcji jest wtedy bezpośrednio wstawiany do programu. Przykład: „funkcja” `getchar()` z biblioteki ANSI C
  - preprocesor nie zna jednak składni języka C/C++ i dlatego nie weryfikuje wielu istotnych rzeczy (np. typów danych)
  - po zdefiniowaniu (dyrektywą) funkcji, **wyglądają** one jak zwykłe funkcje C, **zachowują się** jednak inaczej. Może prowadzić to do błędów

- dyrektywa #define

```
#include <iostream>

#define POCZATEK 3 #define KONIEC 6
#define ROZMIAR KONIEC - POCZATEK

void main() {
    int tab1[ ROZMIAR ];           // OK
    int tab2[ 2 * ROZMIAR ];       // BLAD np: (2*6)-3 zamiast 2*(6-3)
    int i;

    for( i = 0; i < ROZMIAR; i++ ) // OK
      tab1[ i ] = i;

    for( i = 0; i < 2 * ROZMIAR; i++ ) // BLAD
      tab2[ i ] = i;

    cout << "ROZMIAR = " << ROZMIAR << endl;
    cout << "2 * ROZMIAR = " << 2 * ROZMIAR << endl;  }
```


- kwalifikator const
  - problem z poprzedniego slajdu można rozwiązać poprzez zastosowanie **kwalifikatora** `const`
  - w języku **C** użycie `const` sprowadzało się tylko do uniemożliwienia modyfikacji zainicjalizowanej zmiennej. Nie można było używać `const` w miejscu przeznaczonym na stałe symboliczne
  - kompilator C przydzieli zmiennej `const` po prostu pamięć (jak każdej innej zmiennej)
  - kompilator C++ wprowadzi taką **zmienną** do tzw. **tablicy symboli** (oszczędność pamięci przeznaczonej na zmienne robocze)
  - kompilator C oblicza złożone wyrażenie zawierające `const` w trakcie **wykonywania** programu. Nie można by więc zadeklarować `char tab1[ ROZMIAR ];`
  - kompilator C++ oblicza złożone wyrażenie zawierające `const` w trakcie **kompilacji** programu. Można więc zadeklarować `char tab1[ ROZMIAR ];`
  - **Wnioski**:
    - w języku C `const` i `#define` zachowują się inaczej. Nie należy więc stosować ich zamiennie
    - w języku C++ `const` zastępuje z korzyścią `#define`

- kwalifikator const

```
#include <iostream>

const int POCZATEK = 2;                    // Uwaga na znak = oraz ;
const int KONIEC = 6;                      // przy #define ich nie było
const int ROZMIAR = KONIEC - POCZATEK;

void main()
{
    char tab1[ ROZMIAR ];                  // OK
    char tab2[ 2 * ROZMIAR ];              // Teraz OK!
    int i;

    for( i = 0; i < ROZMIAR; i++ )         // OK
      tab1[ i ] = i;

    for( i = 0; i < 2 * ROZMIAR; i++ )     // Teraz OK !
      tab2[ i ] = i;

    cout << "ROZMIAR = " << ROZMIAR << endl;
    cout << "2 * ROZMIAR = " << 2 * ROZMIAR << endl;
}
```


- kwalifikator const
  - kolejne zastosowanie `const`
  - stałe argumenty wskaźnikowe funkcji. Zabezpieczamy je przed (z reguły przypadkową) modyfikacją. Nie dotyczy to argumentów przekazywanych przez wartość. Tam zabezpieczenie to jest "automatyczne" - pracujemy na kopii argumentu
  - kompilator (zarówno C++ jak i ANSI C) nadzoruje dostęp do zmiennej `const`. Próba jej modyfikacji kończy się błędem

- kwalifikator const

```
int strlen(const char* napis);
char* strcpy(char* dest, const char* src); // nie modyfikuj źródła
const char* pobierz_bufor();               // tylko obejrzyj,
                                           // modyfikacja zabroniona

void main()
{
    const int rozmiar = 50;
    int* pi;

    rozmiar = 25;    // BLAD: modyfikacja zmiennej (const int).
    rozmiar++;       // BLAD: jak wyzej.

    pi = &rozmiar;   // BLAD: konwersja (const int*) -> (int*).

    *pi = 25;        //
}
```

konwersja to też modyfikacja !

- funkcje inline a dyrektywa #define
  - dyrektywa `#define` (oprócz definicji stałych symbolicznych) używana jest do definiowania makropoleceń
  - kod makr(opoleceń) wstawiany jest bezpośrednio do kodu wynikowego
  - stosując `#define` zyskujemy na szybkości, gdyż nie ma wtedy potrzeby odkładania (i późniejszego pobierania) funkcji (+ parametry, adresy wywołania, powrotu, itp.) na stos
  - stosując `#define` "tracimy" na wielkości kodu wynikowego. Jeśli wywołujemy makro np. 50 razy, jego kod zostanie 50 razy skopiowany do programu wykonywalnego
  - stosując `#define` mogą pojawić się błędy. Głównie dotyczy to sytuacji, gdy "funkcjom" przekazujemy argumenty i następuje (niezamierzone) wielokrotne obliczanie argumentów

- funkcje inline a dyrektywa #define
  - stosując `inline` "doradzamy" kompilatorowi wygenerowanie kopii kodu funkcji w miejscach wywołań aby uniknąć jej wywoływania
  - kompilator może ignorować `inline` dla funkcji, które uzna że są za duże !
  - `inline` używamy więc tylko z małymi funkcjami i często używanymi
  - `inline` redukuje czas wykonywania, ale zwiększa rozmiar programu

funkcje `inline` będą często używane w klasach

- funkcje inline a dyrektywa #define

```
#include <iostream>

#define ABS( a ) (((a)>0) ? (a) : (-(a)))

void main()
{
    int i = -3;

    cout << "i = " << i << endl;
    cout << "ABS( i ) = " << ABS( i ) << endl;
    cout << "ABS( i++ ) = " << ABS( i++ ) << endl;   // BLAD
    cout << "i = " << i << endl;
}
```

wszędzie w miejsce argumentu `a` podstawiane jest wyrażenie i++. (wykona się wiec ono DWA razy)

```
i = -3
ABS(i) = 3
ABS(i++) = 2
i = -1
```

```
#include <iostream>

inline abs( int a )
{
    if( a > 0 ) return a;
    return -a;
}

void main()
{
    int i = -3;

    cout << "i = " << i << endl;
    cout << "abs( i ) = " << abs( i ) << endl;
    cout << "abs( i++ ) = " << abs( i++ ) << endl; // OK
    cout << "i = " << i << endl;
}
```

```
i = -3
abs(i) = 3
abs(i++) = 2
i = -2
```

- dynamiczny przydział pamięci, operatory new i delete
  - alternatywa dla `malloc() i free()`
  - `new` i `delete` to **operatory**
  - `new` "zwraca" (tak jak `malloc()`) wskaźnik do zaalokowanej pamięci
  - nie ma potrzeby stosowania operatora `sizeof` razem z operatorem `new`. Operator `new` automatycznie wylicza potrzebną liczbę bajtów
  - nie ma potrzeby jawnego rzutowania (jak było w przypadku `malloc()`)
  - `malloc()` w przypadku błędu zwraca **NULL**
    `new` zwraca **0 (tzw. wskaźnik 0)**
  - wskaźnik `0` pełni w C++ taka sama role co stała `NULL` w ANSI C

```
int *Tab new int[10];
if (Tab == 0)
  { OBSLUGA BLEDU }
```


- dynamiczny przydział pamięci, operatory new i delete
  - operator `delete` zwalnia to, co przedzielił `new`
  - działa `jak free()`, jednak nie można używać `delete` (C++) do zwalniania pamięci przydzielonej przez `malloc()` (ANSI C) - efekt jest wtedy nieokreślony
  - składnia operatora `delete` jest inna dla tablic
  - *Do samodzielnego poczytania:*
    Obsługę błędów alokacji pamięci można "zautomatyzować". Zajrzyj do dokumentacji kompilatora i poszukaj opisu funkcji `_set_new_handler()`.
    Inne kompilatory też mają podobne rozwiązania
    `http://h30097.www3.hp.com/cplus/new_handler_3c__std.htm`

- dynamiczny przydział pamięci, operatory new i delete

```
char *ch1, *ch40;

ch1 = (char*)malloc (1);
ch1 = new char;            // dyskusyjna użyteczność ?

ch40 = (char*)malloc (40);
ch40 = new char[40];       // pamięć na łańcuch 40 znaków

double *d1, *d50;

d1 = (double*)malloc (sizeof(double));
d1 = new double;

d50 = (double*)malloc (50 * sizeof(double));
d50 = new double[50];

struct wezel {
   int wartosc;
   struct wezel* lewy;
   struct wezel* prawy;
};
wezel *wezel1 *wezelN; //C++ bez „struct”

wezel1 = (wezel*)malloc (sizeof(wezel));
wezel1 = new wezel;        // pamięć na jeden węzeł

int N;
cout << ”podaj liczbe wezlow: „;
cin >> N;

wezelN = (wezel*)malloc (N * sizeof(wezel));
wezelN = new wezel[N];     // pamięć dla N węzłów
```

operator `sizeof` to operator czasu kompilacji, a nie funkcja czasu wykonania. Z tego powodu wykorzystywanie go nie ma wpływu na prędkość działania programu !

- dynamiczny przydział pamięci, operatory new i delete

```
void main()
{
    double* d;
    d = new double;
    *d = 12.34;

    delete d;                       // Zwolnienie pamięci przydzielonej
                                    // jednemu obiektowi.

    d = new double[ 10 ]; // Przydzielenie pamięci dla tablicy obiektów.
    for( int i = 0; i < 10; i++ )
      d[ i ] = double( i )          // rzutowanie w notacji funkcyjnej
    delete [] d;                    // Zwolnienie pamięci przydzielonej
                                    // dla tablicy obiektów.
}
```

dla czytelności zaleca się stosowanie DWÓCH spacji wokół **[]**

operator `delete` sam określi ile zwolnić pamięci (`new` i `delete` automatycznie śledzą tą informację)

Operatory `new` i `delete` zostały wprowadzone aby prawidłowo zarządzać klasami. Pozwalają one na przykład na zainicjalizowanie obiektu należącego do danej klasy dopiero PO dynamicznym przydzieleniu pamięci dla tego obiektu. Będziemy o tym mówić !!!

- referencje (typ referencyjny)
  - pozwala przekazać do funkcji adres argumentu, używając składni przekazywania przez wartość
  - deklaracje:

```
int i=3, j=2;
int* pi;
pi = &i;      // inicjalizacja wskaźnika
int& ri = i; // inicjalizacja referencji (obowiązkowa)
```

  - w przeciwieństwie do wskaźnika, zmienna referencyjna MUSI zostać zainicjalizowana w momencie deklaracji
  - po zadeklarowaniu referencji, powstaje synonim (alias) zmiennej. Cokolwiek wykonamy na referencji (synonimie), wykonuje się na obiekcie, na który wskazuje referencja
  - wskaźnik `pi` (jego wartość) można, jak każdą zmienną modyfikować - referencji **NIE.** Raz zainicjalizowana, nie może już być zmieniana (nie można jej powiązać z inną zmienną)

```
(*pi)++;  // OK: i = 4
ri++;     // OK: i = 5, inkrementacja zmiennej i z użyciem aliasu
pi = &j;  // OK
ri = &j;  // BLAD: i = &j ???
```


- referencje (typ referencyjny)
  - istota referencji nie sprowadza się tylko do możliwości deklaracji referencji - z tego nie ma wyraźnych korzyści
  - powszechnie używa się ich z funkcjami. Upraszczają one składnię **wywołania** funkcji

```
#include <iostream>

struct zespolona {
    double r, i;
};

void wyswietlaj1( zespolona c ){          // Przekazanie przez wartosc.
    cout << c.r << " + i*" << c.i << endl;
}

void wyswietlaj2( const zespolona* pc ){  // Przekazanie przez wskaznik.
    cout << pc->r << " + i*" << pc->i << endl;
}

void wyswietlaj3( const zespolona& rc ){  // Przekazanie przez referencje.
    cout << rc.r << " + i*" << rc.i << endl;
}

void main() {
    zespolona c;

    c.r = 1.0;
    c.i = -1.0;

    wyswietlaj1( c );
    wyswietlaj2( &c );
    wyswietlaj3( c );
}
```

taka sama treść funkcji !

- referencje (typ referencyjny)
  - referencje można też zwracać jako wartości funkcyjne
  - w praktyce nie stosuje się tego. Mechanizm ten ma praktyczne zastosowanie dopiero przy np. przeciążaniu operatorów klas (C++)

```
int glob = 3;         // Zmienna globalna.

int& refglob()
{
    return glob;
}

void main()
{
    int i;

    i = refglob();    // i = 3.

    refglob() = 7;    // teraz glob = 7. Przypisanie nowej wartości
                      // obiektowi zwracanemu przez funkcję
                      // Co to za skladnia !!!

    i = refglob();    // i = 7.
}
```

Jeżeli funkcja zwraca powiązanie z obiektem (referencję), to istnieje możliwość jego zmodyfikowania przez przypisanie nowej wartości obiektowi zwracanemu przez funkcję

- referencje (typ referencyjny)
  - implementacja referencji
  - referencję do zmiennej implementuje się jako stałe wskazanie, które zostanie automatyczne usunięte przez kompilator

```
int i = 3;

int& ri = i  // Równoważne: int *const pi = &i;
ri = ri + 5; // Równoważne: (*pi) = (*pi) + 5;
ri++;        // Równoważne: (*pi)++;


// referncja do stałej wymaga wcześniejszego utworzenia
// zmiennej pomocniczej

const int& ri = 5; // Równoważne: const int tmp = 3;
                   //              const int *const pi = &tmp;
```

- referencje (typ referencyjny)
  - aby nie było nieporozumień (słusznych !), nie należy dopuszczać do modyfikacji argumentów przekazywanych przez referencję. Do tego służą wskaźniki !
  - jeśli funkcja modyfikuje argumenty - stosuj jawne przekazanie przez wskaźnik
  - jeśli koniecznie "uprzemy" się na referencje - używajmy `const` `void fun (const int& ri){...}`

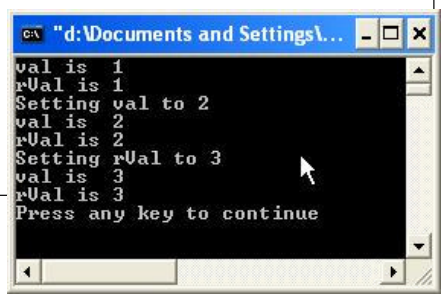
```
int oblicz2( int* pa ) {
    *pa *= 2;
    return (*pa) * (*pa);
}

int oblicz3( int& ra ) { // po ew. const kompilator nie pozwoli na ra *= 2;
    ra *= 2;            // PULAPKA: modyfikowanie zmiennej referencyjnej.
    return ra * ra;
}

void main() {
    int i = 2, j2, j3;


    j2 = oblicz2( &i );
         // OK : zmienna i zmodyfikowana, lecz ryzyko
         // to jest od razu widoczne, gdyz argument
         // przekazywany jest przez adres. (i = 4)

    j3 = oblicz3( i );
         // UWAGA: zmienna i zmodyfikowana,
         // lecz nie jest to widoczne w wywolaniu funkcji. (i = 8) }
```


- referencje (typ referencyjny)
  - Uwagi praktyczne:
  - modyfikowalne argumenty funkcji przesyłaj przez wskaźnik
  - małe NIE-modyfikowalne argumenty funkcji przesyłaj przez wartość
  - duże NIE-modyfikowalne argumenty funkcji przesyłaj przez referencję do stałych
    np.: `void fun (const int& aRef)`

- referencje (typ referencyjny)

```
// inny przykład pokazujący zasadę

#include <iostream>
using namespace std;

int main()
{

    int val = 1;
    int &rVal = val;

    cout << "val is " << val << endl;
    cout << "rVal is " << rVal << endl;

    cout << "Setting val to 2" << endl;
    val = 2;

    cout << "val is " << val << endl;
    cout << "rVal is " << rVal << endl;

    cout << "Setting rVal to 3" << endl;
    rVal = 3;

    cout << "val is " << val << endl;
    cout << "rVal is " << rVal << endl;

    return 0;
}
```

raz modyfikujemy zmienna, a raz referencję do niej



- wartości domyślne argumentów funkcji
  - deklaracja wartości domyślnych argumentów już na etapie deklaracji prototypu funkcji
  - w wywołaniu nie wolno pomijać argumentu (czyli używać wartości domyślnych), jeżeli po prawej stronie argument określony jest jawnie (**BLAD 1**)

```
#include <iostream>

// Zwróć uwagę na wartości domyślne:
void petla( int poczatek = 1, int koniec = 10, int krok = 1)
{
    cout << "--- Petla od " << poczatek << " do " << koniec
    << " z krokiem co " << krok << " ---" << endl;

    for( int i = poczatek; i <= koniec; i += krok )
    cout << "i = " << i << endl;
}

void main()
{
    petla( 15, -10, -5 );    // OK
    petla( 10, 15 );         // OK : petla( 10, 15, 1 );
    petla( 5 );              // OK : petla( 5, 10, 1 );
    petla();                 // OK : petla( 1, 10, 1 );

    // błędy na etapie wywołania funkcji

    petla( -10, ,2 );        // BLAD 1
    petla( , ,2 );           // BLAD
    petla( , 33, 2 );        // BLAD
    petla( , 44, 2 );        // BLAD
}
```

- wartości domyślne argumentów funkcji
  - w deklaracji funkcji wartości domyślne mogą być przypisane tylko argumentom znajdującym się najbardziej z prawej strony listy parametrów

```
#include <iostream>

.
.
.
// błędy na etapie deklaracji funkcji

void petla( int poczatek = 1, int koniec, int krok);          // BLAD
void petla( int poczatek = 1, int koniec, int krok = 1);      // BLAD
void petla( int poczatek = 1, int koniec = 10, int krok);     // BLAD
.
.
.
void petla( int poczatek = 1, int koniec = 10, int krok = 1); // OK
void petla( int poczatek, int koniec = 10, int krok = 1);      // OK
void petla( int poczatek, int koniec, int krok = 1);           // OK
```


- przeciążanie funkcji
  - bardzo użyteczny mechanizm !
  - w ANSI C można było używać mechanizmu "pseudo-przciążania", deklarując funkcję z kwalifikatorem `static`, ograniczając w ten sposób zakres (widoczność) funkcji do pliku, w którym znajdowała się jej deklaracja
  - o funkcji mówimy, że jest przeciążona, gdy pod jedna nazwą umieszczono kilka różnych jej implementacji
  - kompilator określa, którą funkcję wywołać w zależności od typu (ilości) argumentów, z którymi ją wywołano. Aby było to możliwe, tworzona jest tzw. sygnatura funkcji
  - W celu utworzenia sygnatury funkcji, NIE brane są pod uwagę wartości zwracane przez funkcję ! Uwaga też na argumenty referencyjne

```
void fun (int);
void fun (int&); // BLAD. Kompilator nie odróżni (int) od
(int&)
int fun (int);   // BLAD. Kompilator nie odróżni tej
funkcji
                 // od void fun (int)
```

  - mechanizmu funkcji przeciążonych nie należy nadużywać. Jeśli przeciążamy dwie (lub więcej) funkcji, to muszą to być funkcje, które rzeczywiście wykonują to samo zadanie ale na innych typach danych

- przeciążanie funkcji

bez mechanizmu przeciążania

```
#include <iostream>

void wyswietlaj_int( int i ){
    cout << "(int) = " << i << endl;
}

void wyswietlaj_double( double d ){
    cout << "(double) = " << d << endl;
}

const int rozmiar = 10;
typedef long tablica[ rozmiar ];

void wyswietlaj_tablica( tablica t )
{
    cout << "(tablica) = " << endl;
    for( int i = 0; i < rozmiar; i++ )
    cout << "[ " << i << " ] = " << t[ i ] << endl;
}

void main()
{
    int i = 3;
    wyswietlaj_int( i );

    double d = 123.456;
    wyswietlaj_double( d );

    tablica t;
    for( int j = 0; j < rozmiar; j++ )
    t[j] = j*j;
    wyswietlaj_tablica( t );
}
```


- przeciążanie funkcji

z mechanizmem przeciążania

```
#include <iostream>

void wyswietlaj( int i ){
    cout << "(int) = " << i << endl;
}

void wyswietlaj( double d ){
    cout << "(double) = " << d << endl;
}

const int rozmiar = 10;
typedef long tablica[ rozmiar ];

void wyswietlaj( tablica t )
{
    cout << "(tablica) = " << endl;
    for( int i = 0; i < rozmiar; i++ )
    cout << "[ " << i << " ] = " << t[ i ] << endl;
}

void main()
{
    int i = 3;
    wyswietlaj( i );

    double d = 123.456;
    wyswietlaj( d );

    tablica t;
    for( int j = 0; j < rozmiar; j++ )
    t[ j ] = j*j;
    wyswietlaj( t );
}
```

- wzorce funkcji (szablony funkcji)
  - mechanizm intensywnie wykorzystywane w bibliotekach obiektowych (funkcje wzorcowe w klasach, klasy szblonowe). O tym będzie jednak później
  - funkcje przeciążone są zwykle stosowane do przeprowadzania podobnych działań wymagających różnej logiki ale na różnych typach danych
  - jeżeli logika ta jest identyczna dla każdego typu danych, wygodnie jest używać wzorców funkcji
  - Jak to zrobić ?
  - programista pisze pojedynczą definicję wzorca funkcji o sparametryzowanym typie argumentu
  - na bazie typów argumentów dostarczonych w wywołaniu tej funkcji, kompilator automatycznie generuje oddzielne funkcje wzorcowe. Używa się tez pojęcia konkretyzacja funkcji

Szablony klas. Bardzo potężny mechanizm w języku C++. Będziemy o tym mówić !!!

- wzorce funkcji (szablony funkcji)

```
#include <iostream>

// Dla typu int.
int min( int a, int b ){
    if( a <= b ) return a;  return b;}

// Dla typu long.
long min( long a, long b ){
    if( a <= b )  return a; return b;}

// Dla typu char.
char min( char a, char b ){
    if( a <= b )  return a; return b;}

// Dla typu double.
double min( double a, double b ){
    if( a <= b )  return a; return b;}

void main()
{
    int ia = 1, ib = 5;
    cout << "(int) : " << min( ia, ib ) << endl;

    long la = 10000, lb = 5000;
    cout << "(long) : " << min( la, lb ) << endl;

    char ca = 'a', cb = 'z';
    cout << "(char) : " << min( ca, cb ) << endl;

    double da = 123.456, db = 789.10;
    cout << "(double) : " << min( da, db ) << endl;
}
```

"oszczędne" formatowanie kodu źródłowego.
Prosimy nie powtarzać ! :-(

Wiele przeciążonych funkcji o identycznym kodzie

- wzorce funkcji (szablony funkcji)

```
#include <iostream>

template <class T>
T min( T a, T b )
{
    if( a <= b )
        return a;
    return b;
}


void main()
{
    int ia = 1, ib = 5;
    cout << "(int) : " << min( ia, ib ) << endl;

    long la = 10000, lb = 5000;
    cout << "(long) : " << min( la, lb ) << endl;

    char ca = 'a', cb = 'z';
    cout << "(char) : " << min( ca, cb ) << endl;

    double da = 123.456, db = 789.10;
    cout << "(double) : " << min( da, db ) << endl;
}
```

Tylko jeden wzorzec

Wzorzec ten deklaruje jeden parametr formalnego typu T

- wzorce funkcji (szablony funkcji)

```
// inne przykłady szablonów

// Prototype (Declaration):
void ConvertFToC (float f, float &c);

// Definition:
void ConvertFToC (float f, float &c)
{
    c = (f - 32.) * 5./9.;
}

// porównaj z:

template <class Type>
    void ConvertFToC (Type f, Type &c);

template <class Type>
    void ConvertFToC (Type f, Type &c)
{
    c = (f - 32.) * 5./9.;
}
```

– Kompilator w ogólności stara się "wydedukować" typy parametrów z postaci wywołania funkcji i z reguły poprawnie konkretyzuje funkcje (ang. template instantiation)

- wzorce funkcji (szablony funkcji)

```
// inne przykłady szablonów

template <class type1, class type2>
    type1 SomeFunction(type1 a, type2 b, type2 c);

int result;
int k = 1;
float y = 4.5, z = 3.9;

....
result = SomeFunction (k, y, z);
```

– zdarza się jednak, że typ argumentu należy podać jawnie. Wywołanie funkcji wygląda wtedy nastepująco

```
result = SomeFunction<int, float>(k, y, z);
```

– słowo kluczowe `typename` może być wymiennie używane z `class`

```
template <typename type1, typename type2>
    type1 SomeFunction (type1 a, type2 b, type2 c);
```


- wzorce funkcji (szablony funkcji)

– jeżeli dla kilku argumentów wzorca funkcji wyspecyfikowany został ten sam typ, wtedy parametry przekazywane w wywołaniu funkcji muszą być również jednakowego typu

```
    String s1("Napis");           // "jakiś" typ String
    double da = 123.456;

    cout << "(???) : " << min( da, s1 ) << endl;

    // BLAD
    // nie można skonkretyzować funkcji min(double, String)
```

– można jednak używać argumentów różnych typów (np. dwa parametry sparametryzowane, inne jawnie określone)

```
template <class T> double fun( int a, T b, T c, double d )
{
    T temp = b + c;
    return a + d * temp;
}

void main()
{
 int b1 = 1, c1 = 7;
 cout << fun(8, b1, c1, 5.3);

 double b2 = 1.23, c2 = 7.3665;
 cout << fun(8, b2, c2, 5.3);

 String s1("Napis1"); String s2("Napis2");
 cout << fun(8, s1, s2, 235.3);          //BLAD. Ile wynosi a + d * temp?
```

- wzorce funkcji (szablony funkcji)
  - można używać argumentów sparametryzowanych różnych typów

```
template <class T class U> U fun( T a, U b )
{
    return a + b;
}

void main()
{
 fun (19, 44);            // konkretyzacja fun (int, int)
 fun (23.55, 45);         // konkretyzacja fun (double, int)
 fun (78, "Napis");       // konkretyzacja fun (int, char*)

 int Tab[10];
 fun (3.54, Tab);         // konkretyzacja fun (double, int*)
```

  - oczywiście w zależności od zawartości funkcji `fun`, nie wszystkie konkretyzacje zostaną przyjęte przez kompilator
  - przykładowe wywołania powyżej pokazują tylko jak kompilator będzie chciał konkretyzować poszczególne wywołania
  - przykład: jak dodać do siebie `double i int*` ?

- jednoargumentowy operator ::
  - powszechnie używany z klasami (to "wiedzą wszyscy"). Tam występuje jednak jako operator dwuargumentowy
  - w wersji jednoargumentowej używany do rozróżniania zasięgu lokalnych i globalnych zmiennych o takiej samej nazwie
  - dwie zmienne PI (globalna i lokalna) są innego typu tylko dla czytelności przykładu

```
#include <iostream>

const double PI = 3.14159265358979;

int main()
{
  const float PI = static_cast< float >( ::PI );

  cout << setprecision(20)
       << "Lokalna wartosc typu float liczby PI = " << PI
       << "Globalna wartosc typu double liczby PI = " << ::PI << endl;

  return 0;
}
_____________
Wynik:
Lokalna wartosc typu float liczby PI = 3.14159
Globalna wartosc typu double liczby PI = 3.14159265358979
```

należy unikać używania zmiennych o tej samej nazwie dla różnych celów w programie. Praktyka taka prawie zawsze prowadzi do kłopotów !

- korzystanie z bibliotek funkcji C w programach C++
  - często pojawi się potrzeba użycia kodu napisanego w C w programie C++
  - kompilator C++ koduje nazwy funkcji przed przesłaniem ich do konsolidatora. Miedzy innymi dlatego poprawnie "działa" mechanizm przeciążania funkcji.
  - "zakodowana" nazwa oprócz właściwej nazwy funkcji, zawiera też w nazwie typy argumentów funkcji kodowanej
  - różne kompilatory różnie kodują nazwy. Nie ma tu "ogólnoświatowej" zgodności, gdyż programista nie musi tego znać (chyba, że analizuje kod assemblera wygenerowany przez kompilator)
  - język ANSI C NIE koduje nazw funkcji i tu leży problem
  - dyrektywa `extern "C"` rozwiązuje problem. Zawiesza ona kodowanie funkcji i dlatego unikamy błędów, które pojawiły by się na etapie konsolidacji programu, który korzysta w C++ z kodu C. Poniżej trzy przykłady jej użycia

```
extern "C"
{
#include "moje.h"   // kompilowane w ANSI C
}

void main()
{
  moje_fun ();
}
```

```
#ifdef _cplusplus
extern "C"{
#endif

... deklaracje funkcji

#ifdef _cplusplus
}
#endif
```

```
extern "C" void fun(double);
```


- wskaźnik NULL i wskaźnik 0
  - w C++ wszystkie zerowe wartości kodowane są jako 0 (zero)
  - w C istnieje wyjątek. Przy pracy ze wskaźnikami używa się wartości NULL
  - w C++ zaleca się używanie wartości 0 również do wskaźników pustych (choć NULL jest w dalszym ciągu dostępne)
  - wskaźnik **0** pełni w C++ taka sama role co stała NULL w ANSI C

```
int *Tab new int[10];
if (Tab == 0)
  { OBSLUGA BLEDU }
```

- #define __cplusplus
  - każdy kompilator C++ pretendujący do miana zgodnego z ANSI/ISO definiuje symbol `__cplusplus`
  - każdy plik źródłowy zachowuje się więc tak, jakby posiadał dyrektywę `#define __cplusplus`.

- funkcje elementem struktur
  - w C++ zezwala się na definiowanie funkcji jako części struktur

```
struct point
 {
   int x;
   int y;
   void draw(void);
 };

point a;
point b;

a.x = 0;
a.y = 10;
a.draw();

b = a;
b.y = 20;
b.draw();
```

a gdzie definicja funkcji ?

```
struct Person
 {
   char name[80],
   char address[80];
   void print();
 };

void Person::print()
  {
    cout << "Name:      " << name << endl
            "Address:   " << address<< endl;
  }

Person p;
strcpy(p.name, "xyz");
strcpy(p.address, "abc");
p.print();
```

analogicznie będzie przy definicji metod będących elementem klas

- nowe typy danych

```
bool

typ całkowity, który może przyjmować wartości true lub false.
Rozmiar typu nie jest określony

// relacje pomiędzy wartościami
!false == true
!true == false

wchar_t

The wchar_t type is an extension of the char basic type, to accomodate wide
character values, such as the Unicode character set. The  g++ compiler
(version 2.95 or beyond) reports sizeof(wchar_t) as 4, which easily
accomodates all 65,536 different Unicode character values.
```

- operatory
  - w C zbiór operatorów jest podzbiorem operatorów w C++
  - przykłady operatorów specyficznych dla C++ (`this`, `delete`, ...)

```
   and        const     float         operator  static_cast    using
and_eq   const_cast       for               or       struct  virtual
   asm     continue    friend            or_eq       switch     void
  auto      default      goto          private     template volatile
bitand       delete        if        protected         this  wchar_t
 bitor           do    inline           public        throw    while
  bool       double       int         register         true      xor
 break dynamic_cast      long reinterpret_cast          try   xor_eq
  case         else   mutable           return      typedef
 catch         enum namespace            short       typeid
  char     explicit       new           signed     typename
 class       extern       not           sizeof        union
 compl        false    not_eq           static     unsigned
```

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

symboliczne alternatywy dla:

`&&, &=, &, |, ~, !, !=, ||, |=, ^, ^=`

- nagłówki
  - These 52 C++ library headers (with the additional 18 Standard C headers) constitute a hosted implementation of the C++ library

nowe nazewnictwo nagłówków ANSI C

```
<algorithm>     <bitset>        <cassert>
<cctype>        <cerrno>        <cfloat>
<ciso646>       <climits>       <clocale>
<cmath>         <complex>       <csetjmp>
<csignal>       <cstdarg>       <cstddef>
<cstdio>        <cstdlib>       <cstring>
<ctime>         <cwchar>        <cwctype>
<deque>         <exception>     <fstream>
<functional>    <hash_map>      <hash_set>
<iomanip>       <ios>           <iosfwd>
<iostream>      <istream>       <iterator>
<limits>        <list>          <locale>
<map>           <memory>        <new>
<numeric>       <ostream>       <queue>
<set>           <sstream>       <stack>
<stdexcept>     <streambuf>     <string>
<strstream>     <utility>       <valarray>
```

na podstawie: `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcstdlib/html/vclrfcplusplusliblaryoverview.asp`