Programowanie w języku C - Daniel Kaczmarski, Adam Krechowicz

1 Język C++

Język C++ jest językiem ściśle wywodzącym się z języka C. Wiele konstrukcji znanych z języka C działa identycznie w C++. W stosunku do C język C++ został rozszerzony m. in. o następujące elementy:

- Obiektowość
- Przestrzenie nazw
- Bibliotekę standardową

Kody źródłowe języka C++ zwykle mają rozszerzenie .cpp (rzadziej .cxx). Pliki nagłówkowe mogą mieć rozszerzenie .h, .hpp, .hxx. kompilowania programu napisanego w języku C++ jest podobny do metody kompilowania programu w języku C. Podstawową różnicą jest jednak wykorzystywany kompilator. Zamiast polecenia gcc należy użyć g++.

```
g++ -o test test.cpp
```

2 Dostęp do bibliotek języka C

Język C++ jest w pełni zgodny wstecz z językiem C. Możliwe jest również używanie bibliotek tego języka. Nie zaleca się jednak używania nazw nagłówków z rozszerzeniem .h. W zamian zalecane jest użycie nagłówka bez rozszerzenia i z przedrostkiem „c". Przykłady:

C	C++
#include <stdio.h>	#include <cstdio>
#include <time.h>	#include <ctime>
#include <stdlib.h>	#include <cstdlib>

3 Operacje wejścia-wyjścia

Operacje wejścia-wyjścia nie są częścią języka C++. Ich wykonywanie umożliwiają biblioteki standardowo dołączane przez producenta kompilatora. Wprowadzanie i wyprowadzanie informacji w języku C++ zostało zrealizowane za pomocą strumieni. Wszystkie strumienie są obiektami odpowiednich klas.

Wykorzystanie w programie strumieni wymaga dołączenia pliku nagłówkowego iostream:

```
#include <iostream>
```

W języku C++ dostępne są cztery predefiniowane strumienie:

- cout – związany ze standardowym urządzeniem wyjścia (ekran),
- cin – związany ze standardowym urządzeniem wejścia (klawiatura),
- cerr – związany ze standardowym urządzeniem, na które chce się wypisywać komunikaty o błędach (standardowo ekran) – strumień niebuforowany,
- clog – związany ze standardowym urządzeniem, na które chce się wypisywać komunikaty o błędach standardowo ekran) – strumień buforowany.

Za wysyłanie i odbieranie informacji ze strumienia odpowiadają operatory « i »:

- « operator odpowiadający za wysyłanie informacji do strumienia, nazywany jest często operatorem wstawiania,
- » operator odpowiadający za wczytywanie informacji, nazywany jest operatorem ekstrakcji.

```
int x=10;
float y=10.218;
cout << x << "\n" << y << endl;
```

Ogólne zasady dotyczące wyświetlania danych przy zastosowaniu strumienia cout i operatora « są następujące:

- liczby całkowite wyświetlane są w systemie dziesiętnym;
- zmienne typów char, unsigned char wyświetlane są jako pojedyncze znaki;
- liczby zmiennoprzecinkowe (float, double) wyświetlane są z dokładnością do 6 cyfr (częśćcałkowita i ułamkowa, bez zbędnych zer);

- wskaźniki wyświetlane są w systemie szesnastkowym;
- zmienne typów char *, unsigned char *wyświetlane są jako łańcuchy znaków.

```
char imie[255];

cin >> imie;
```

Ogólne zasady dotyczące wczytywania danych przy zastosowaniu strumienia cin i operatora » są następujące:

- białe znaki (spacja, tabulacja, enter) są ignorowane;
- wczytywanie tekstów jest kończone po napotkaniu pierwszego białego znaku;
- liczby wczytywane są w systemie dziesiętnym;
- nie można umieszczać spacji pomiędzy znakiem liczby a jej wartością;
- wczytywanie liczby całkowitej jest kończone, gdy kolejny znak nie jest cyfrą;
- w liczbach zmiennoprzecinkowych nie może występować spacja w środku.

4 Funkcja

Funkcje umożliwiają podzielenie oraz pogrupowanie często wykonywanego kodu źródłowego (instrukcji programu) na mniejsze moduły (bloki), które mogą być wielokrotnie wykorzystywane w programie. Zadaniem funkcji zazwyczaj jest wykonywanie określonych operacji oraz zwracanie lub też nie określonej wartości. Często ukrywają one w sobie szczegóły pewnych operacji przed częściami programu, w których znajomość tych szczegółów jest zbędna lub niewskazana.

Wykorzystywanie funkcji w programie daje większą przejrzystość kodu oraz łatwiejsze wprowadzanie zmian.

```
#include <stdio.h>
#include <stdlib.h>

int warunek(int a, int b){
  if(a > b){
    return (1);
```

```
  }
  if(a < b){
    return (2);
  }
  return (0);
}

int main(void){
  int x,y;
  printf("Podaj liczbe calkowita \"x\": ");
  scanf("%d",&x);
  printf("Podaj liczbe calkowita \"y\": ");
  scanf("%d",&y);
  if(warunek(x,y) == 1){
    printf("Liczba %d jest wieksza od liczby %d\n",x,y);
  } else if(warunek(x,y) == 2){
    printf("Liczba %d jest mniejsza od liczby %d\n",x,y);
  } else{
    printf("Liczba %d jest rowna liczbie %d\n",x,y);
  }
  return (0);
}
```

4.1 Parametry funkcji

Do funkcji można przekazać parametry

```
int funkcja(int a, int b){
  return a + b;
}
```

4.2 Wartości zwracane

Funkcja może zwracać wartość dowolnego typu. Wartość zwracana określana jest za pomocą słowa kluczowego return. Po wykonaniu instrukcji return funkcja jest zakańczana nawet gdy nie jest ostatnią instrukcją. W przypadku gdy funkcja nie zwraca żadnej wartości w specyfikacji typu używa się słowa kluczowego void.

4.3 Przypisywanie zwracanych wartości

Funkcja może przypisywać zwracaną wartość do zmiennej niekoniecznie tego samego typu

```
#include <stdio.h>
#include <stdlib.h>

char funkcja(){
  return 5;
```

```
}

void main(){
  short s;
  s = funkcja();
}
```

Podczas przypisywania do innego typu następuje rzutowanie. Typ jest niejawnie konwertowany na odpowiedni. Aby określić właściwą konwersję można wykorzystać rzutowanie jawne.

```
#include <stdio.h>
#include <stdlib.h>

char funkcja(){
  return 5;
}

void main(){
  short s;
  s = (short)funkcja();
}
```

4.4 Przeciążanie funkcji

```
#include <iostream>

void funkcja(){
        std::cout << "Bez parametrow" << std::endl;
}

void funkcja(int i){
        std::cout << "Z parametrem " << i << std::endl;
}

void funkcja(char c){
        std::cout << "Z innym parametrem " << c << std::endl;
}

int main(){
        funkcja();
        funkcja(1);
        funkcja(2);
        funkcja('a');
        funkcja('b');
        return 0;
}
```

4.5 Funkcje inline

Funkcje oznaczone słowem kluczowym *inline* nie są wywoływane w standardowy sposób. Kompilator wprowadza zawartość funkcji inline w każydm miejscu jej wywołania przez co nie jest konieczne wykonywanie skoków do

miejsca gdzie zapisana jest funkcja a następnie powrotu z niej. Należy pamiętać, że wykorzystywanie funkcji inline może znacząco zwiększyć rozmiar wynikowego programu ale wykonanie jej jest szybsze niż standardowej funkcji.

```
inline void foo(){
  printf("asdf\n");
}
```

5 Wiele plików z kodem

5.1 Dołączanie plików

Dzięki dyrektywie *include* możliwe jest dołączenie zawartości dodatkowych plików do kodu źródłowego. Jest to przydatna cecha podczas podziału programu na wiele plików z kodem.

W języku C program może być podzielony na wiele plików źródłowych. W plikach *.c zawarte są kody funkcji, natomiast w plikach *.h zawarte są szkielety funkcji.

5.2 Plik1.h

```
int dodawanie(int x);
```

5.3 Plik1.cpp

```
#include "plik1.h"

int ab = 0;

int dodawanie(int x){
  ab+= x;
  return ab;
}
```

5.4 Plik.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include "plik1.h"

void main(){
printf("%d\n", dodawanie(3));
```

```
printf("%d\n", dodawanie(4));
}
```

5.5 Kompilacja

```
g++ plik.cpp plik1.cpp
```

5.6 Prawidłowe pliki h

W przypadku skomplikowanej struktury plików źródłowych często może występować sytuacja, w której pliki h dołączane są do kodu kilka razy. Np plikA.c zawiera dyrektywę include plików tools.h oraz plikB.h. Jeśli plikB.h także zawiera dyrektywę include pliku tools.h to wynikowo plik ten zostałby załączony dwa razy co jest sytuacją niepożądaną. Dlatego często stosuje się zabezpieczenia plików nagłówkowych przed wielokrotnym załączaniem.

```
#ifndef _TOOLS_H_
#define _TOOLS_H_

/* Zawartosc pliku h */

#endif /* _TOOLS_H_ */
```

W powyższym przykładzie wykorzystano warunkowe definicje preprocesora. Pierwsza z nich *ifndef* sprawdza czy podany jej argument nie został już wcześniej zadeklarowany. Jeśli podana definicja nie została jeszcze zadeklarowana to zostaje to wykonane w drugiej linii. W przeciwnym wypadku gdy definicja istnieje plik jest pomijany, gdyż był już wcześniej dołączony. Dyrektywa *endif* kończy warunek rozpoczęty dyrektywą *ifndef*.

6 Makefile

Narzędzie *Makefile* zarządza procesem kompilacji wielu plików

```
all: plik.o plik1.o
  g++ plik.o plik1.o

plik.o: plik.cpp
  g++ -c plik.cpp

plik1.o: plik1.cpp
  g++ -c plik1.cpp

clean:
  rm -f *.o
```

Plik Makefile składa się z reguł kompilacji:

```
<regula>: <zaleznosci>
  <polecenie>
```

Uwaga: Przed poleceniem musi występować TAB

6.1 Reguły

- all – domyślna reguła kompilująca cały kod
- clean – reguła usuwająca pliki pośrednie

6.2 Wywołanie

- make – wywołanie reguły all z pliku Makefile
- make reg – wywołanie reguły reg
- make clean – wywołanie reguły clean

7 Zadania do wykonania

Podczas wykonywania zadań do wyświetlania rezultatów i pobierania danych od użytkownika wykorzystać strumienie cin oraz cout:

1. Zapoznać się z dokumentacją klasy String dostępnej pod adresem `http://www.cplusplus.com/reference/string/string/`
2. Zapoznać się z dokumentacją klasy Vector dostępnej pod adresem `http://www.cplusplus.com/reference/vector/vector/`
3. Przygotować funkcje obliczające pole koła, trapezu, trójkąta
4. Przygotować rekurencyjną funkcję liczącą n-ty wyraz fibonacciego
5. Przygotować rekurencyjną funkcję liczącą silnię
6. Umieścić wcześniej przygotowane funkcje w osobnych plikach źródłowych
7. Przygotować plik makefile dla programu wykorzystującego pliki z poprzedniego zadania
8. Przygotować funkcje liczące średnią dla dwóch, trzech elementów, tablicy elementów, zakresu elementów w tablicy. Wykorzystać przeciążanie funkcji

9. Przetestować działanie metod klasy string

10. Przygotować kolekcję typu vector przechowującą obiekty string

11. Przygotować program wyszukujący podany ciąg we wszystkich łańcuchach w kolekcji vector